Efficient FHEW Bootstrapping with Small Evaluation Keys, and Applications to Threshold Homomorphic Encryption

Yongwoo Lee[1], Daniele Micciancio[2], Andrey Kim[1], Rakyong Choi[1], Maxim Deryabin[1], Jieun Eom[1], and Donghoon Yoo[1]

[1] Samsung Advanced Institute of Technology, Suwon, Republic of Korea
{yw0803.lee, andrey.kim, rakyong.choi, max.deriabin, jieun.eom, say.yoo}@samsung.com
[2] University of California, San Diego, USA
daniele@cs.ucsd.edu

October 11, 2022

Abstract

The FHEW fully homomorphic encryption scheme (Ducas and Micciancio, Eurocrypt 2015) and its TFHE variant (Chillotti et al., Asiacrypt 2016) are the best-known methods to perform bit-level homomorphic computations on encrypted data. There are two competing bootstrapping approaches to FHEW-like schemes: the AP bootstrapping method (Alperin-Sheriff and Peikert, Crypto 2014) which is the basis of the original FHEW scheme, and the GINX bootstrapping method (Gama et al., Eurocrypt 2016), adopted by TFHE. An attractive feature of the AP/FHEW method is that it supports arbitrary secret key distributions, which is critical for a number of important applications (like threshold and some multi-key homomorphic encryption (HE) schemes) and provides potentially better security guarantees. On the other hand, GINX/TFHE bootstrapping uses much smaller evaluation keys, but it is directly applicable only to binary secret keys, which restricts the scheme's applicability. (Extensions of GINX/TFHE bootstrapping to arbitrary keys are known, but they incur a substantial performance penalty.)

In this paper, we present a new bootstrapping procedure for FHEW-like schemes that achieves the best features of both schemes: support for arbitrary secret key distributions at no additional runtime costs, while using small evaluation keys. As an added benefit, our new bootstrapping procedure results in smaller noise growth than both AP and GINX, regardless of the key distribution.

Our improvements are both theoretically significant (offering asymptotic savings, up to a $O(\log n)$ multiplicative factor, either on the running time or public evaluation key size), and practically relevant. For example, for a concrete 128-bit target security level, we show how to decrease the evaluation key size of the best previously known scheme by more than 30%, while also slightly reducing the running time. We demonstrate the practicality of the proposed methods by building a prototype implementation within the PALISADE open-source HE library. We provide optimized parameter sets and implementation results showing that the proposed algorithm has the best performance among all known FHEW bootstrapping methods in terms of runtime and key size. We illustrate the benefits of our method by sketching a simple construction of threshold HE based on FHEW.

Keywords: Blind Rotation, Bootstrapping, Fully Homomorphic Encryption (FHE), Threshold Homomorphic Encryption.

Contents

1 Introduction

There are two competing approaches to bootstrap FHEW-like Fully Homomorphic Encryption (FHE) schemes [DM15, CGGI20]: the AP bootstrapping method (originally proposed by Alperin-Sheriff and Peikert [AP14] and efficiently instantiated in the ring setting by the FHEW cryptosystem [DM15]), and the GINX method (originally proposed by Gama et al. [GINX16] and adapted to the ring setting by the TFHE scheme [CGGI20].) A detailed comparison between the two methods is presented in [MP21], which concludes that the AP/FHEW method [AP14, DM15] is faster when LWE (Learning With Errors) secret keys follow the Gaussian distribution, while the (ring) GINX/TFHE method [GINX16, CGGI20] has the lead for the special case of *binary* LWE secret keys. For the crossover point of ternary keys, [MP21] still recommends GINX bootstrapping due to its much lower memory (bootstrapping key) requirements. Very recently, [KDE$^+$21, BIP$^+$22] further improved GINX bootstrapping for ternary secrets by reducing the computation by half. (In our comparison, we refer to this optimized scheme as GINX*.) Besides the smaller running times, a big attraction of using GINX bootstrapping (with binary or ternary keys, as implemented by [CGGI20, MP21]) is its lower memory footprint, which is substantially smaller than the AP method.

Efficiency aside, the use of secret keys with large entries is still interesting for both theoretical and practical reasons. On the theoretical side, the foundation of lattice cryptography only offers solid support for Gaussian keys with relatively large entries, of the order of $O(\sqrt{n})$ [Reg09, LPR13], where n is the secret vector dimension serving as a security parameter. The use of smaller keys (e.g., with binary coefficients) has also received a substantial amount of theoretical attention [MP13, Mic18, GKPV10, BLP$^+$13, BD20, KF15]. However, the current state of the art, provided by [Mic18][1], only shows that LWE with binary secrets can be proved as hard as standard LWE (with uniform or gaussian secrets) at the cost of increasing the secret dimension by a factor $O(\log q)$ and the error rate by a factor $O(\sqrt{n})$. So, motivated by practical considerations (limiting error growth during homomorphic computation, and the efficient implementation of GINX bootstrapping), these theoretical results supporting binary secrets are typically ignored, and parameters are set based on the best currently known attacks.[2] For fairness, this is also the approach followed in this paper when comparing our work to previous schemes that benefit from the use of binary secrets.

A more compelling motivation to use larger secret keys in practice is offered by *threshold* (lattice-based, homomorphic) encryption [BD10, AJLA$^+$12]. Threshold cryptography offers a method to distribute a secret key s among a set of participants, say $P_1, \dots, P_k$, each holding a share s_i of the secret key, in such a way that they can collaboratively decrypt messages. Still, if a subset of parties are corrupted and their secret shares s_i are made available to an adversary, ciphertexts retain their security. So, threshold cryptography eliminates the single point of failure associated with the secret key and ensures that encrypted data remains secure unless collaboratively decrypted by all parties.

The use of threshold cryptography is particularly attractive in the setting of homomorphic computation, as it requires modifications only to the key generation and decryption procedures. A threshold encryption scheme still has a *single public key* p (under which all messages can be

[1] The more recent work [BD20] provides more general results for arbitrary "entropic" distributions, but does not improve the reduction for binary secrets.

[2] This has become quite common for uniformly random binary secrets, and their use is now included in practical tools, like the lattice estimator of [APS15]. Other, for extreme parameter settings (e.g., sparse keys or a very large number of samples), there remain concerns about weakening the security of the LWE problem [AG11, CHHS19], and their use is generally discouraged.

encrypted by different parties) and *evaluation key* (used to perform homomorphic computations on ciphertexts.) In other words, applications of threshold Homomorphic Encryption (HE) support the same, simple workflow of standard (single party) HE: all data owners encrypt their data under a single public key p, and send their encrypted data to a single server that securely performs the encrypted computation, leading to a final encrypted result. Only at this point, the protocol requires interaction with multiple decryption servers (each holding a secret share s_i) to recover the final result. So, by only increasing the cost of decryption (and only by a modest amount, see below), threshold cryptography guarantees the security of all data (encrypted under a common public key p), even against the servers holding the decryption key (as long as they are not all corrupted.[3])

Lattices (and the LWE problem) provide a very convenient setting to implement threshold cryptography, as the public key ($p \approx a \cdot s$) is defined as a (noisy) linear function of the secret key s, for a random, publicly known value a. (See next section for a more formal definition of the LWE function.) So, distributed (shared) key generation can be easily implemented by having each party choose a local public-secret key pair ($p_i \approx a \cdot s_i, s_i$) individually (without any interaction), and then setting the public key to the sum $p = p_1 + \cdots + p_k$ of the local public keys. It is immediate to see that this is a valid public key corresponding to the secret key $s = s_1 + \cdots + s_k$ implicitly shared by all parties. In fact, this is how keys are generated in [BD10] (using uniformly random s_i) and [AJLA$^+$12] (using an arbitrary LWE key generation algorithm for each s_i.) Decryption can also easily be implemented[4] by decrypting a ciphertext c using the individual secret key shares and then adding up the partial decryptions. So, key generation and decryption are minimally interactive.[5] We remark that the threshold schemes [BD10, AJLA$^+$12] predate FHEW-like HE ([BD10] does not explicitly provide any homomorphic computation capability, and [AJLA$^+$12] is a BGV-type encryption scheme.) However, the same principles apply to virtually any LWE-based encryption scheme, including those considered in this paper. Now comes a critical observation: even if the local key shares s_i have binary coefficients, their sum s (used by homomorphic computations and bootstrapping) is no longer binary and has coefficients potentially as large as k, the number of parties participating in the shared decryption protocol. Depending on the application, this number can be quite high, requiring similarly large secret keys. (E.g., see [CHI$^+$21] for an application of lattice-based threshold (additively) HE with as many as 1000 parties.)

The FHEW-like cryptosystems with either AP or GINX bootstrapping are the most attractive methods for bit-level homomorphic computations.[6] But when ported to the threshold cryptography setting (with its correspondingly larger secret keys), FHEW-like encryption presents the user with a difficult choice between

- the AP bootstrapping method of [AP14, DM15], with its fast performance (essentially independent of the secret key size) but very large evaluation keys, and
- the GINX bootstrapping method and its variants [GINX16, CGGI20, MP21, JP22], with

[3]As standard in HE, we consider security against passive adversaries, in which case the security threshold can be set to $k - 1$.

[4]This requires some care, adding noise to the partial decryptions to avoid information leakage, as already done in [BD10, AJLA$^+$12]. In this paper, we focus on the distributed key generation and homomorphic computation stages, which are the most relevant to FHE bootstrapping.

[5]HE also requires the generation of public evaluation keys, which introduces some additional complications, and is discussed below.

[6]Other methods oriented towards arithmetics on integer or approximations to real/complex numbers like [BGV14, CKKS17] offer advantages for a complementary set of applications, but are not within the scope of our paper.

much smaller evaluation keys, but substantially larger running time due to the use of large secret keys.

A related class of applications to "multi-key HE" is discussed later on. So, one may ask the question: is it possible to design a bootstrapping procedure that offers the advantages of both methods, i.e., fast bootstrapping with arbitrarily distributed secret keys and small public evaluation keys?

1.1 Our results

We answer the above question in the affirmative, designing a new bootstrapping procedure that supports the use of arbitrary secret key distributions without any performance penalty (similar to AP/FHEW bootstrapping) while keeping the attractive small size of GINX/TFHE bootstrapping keys. In fact, we even improve upon the performance of the best previously known scheme (with binary secrets), both in terms of key size and running time. For example, for the simple case of a (single user) gate bootstrapping operation at a 128-bit target security level, we improve upon previous schemes by reducing the evaluation key size by 30% (from 20MB to 14MB) while also slightly reducing the running time. (See Section 5 for details.) The impact of our bootstrapping method becomes significant with larger keys or a moderately large number of threshold decryption servers. Our method offers the additional advantage of reducing the amount of noise introduced during bootstrapping, even for the case of binary keys that are the most favorable to GINX so far. The improvements over previous methods are both theoretical (reducing either the running time or memory requirement of previous bootstrapping procedure by factors as high as $O(\log n)$, depending on the size of the secret keys/threshold group size), and practical. We verified our theoretical results and the practicality of the proposed method with experiments, performed using prototype implementation within the PALISADE open-source HE library.

1.2 Techniques

The main operation underlying both AP and GINX bootstrapping is the evaluation of a so-called "blind rotation". This operation takes some polynomial $\boldsymbol{f}_0$ as an input and "rotates" it by some value encrypted within a given LWE ciphertext $(\vec{\alpha} = (\alpha_0, \dots, \alpha_{n-1}), \beta)$ using secret key $\vec{s} = (s_0, \dots, s_{n-1})$. (See Section 2 for more details.)

Starting with the encryption $\mathsf{RLWE}(\boldsymbol{f}_0)$ of a polynomial $\boldsymbol{f}_0$ previous blind rotation algorithms work as follows: at step i, given an encryption $\mathsf{RLWE}(\boldsymbol{f}_{i-1})$ of a polynomial $\boldsymbol{f}_{i-1}(X) = \boldsymbol{f}_0 \cdot X^{\sum_{j \le i-1} \alpha_j s_j}$, homomorphically compute an encryption $\mathsf{RLWE}(\boldsymbol{f}_i)$ of an updated polynomial $\boldsymbol{f}_i = \boldsymbol{f}_{i-1} \cdot X^{\alpha_i \cdot s_i} = \boldsymbol{f}_0 \cdot X^{\sum_{j \le i} \alpha_j s_j}$, using a publicly known constant α_i (part of the input LWE ciphertext) and an encryption[7] $E(s_i)$ of a secret key coordinate s_i. After repeating this step n times, we obtain the encryption of $\mathsf{RLWE}(\boldsymbol{f}_0 \cdot X^{\langle \vec{\alpha}, \vec{s} \rangle})$, which is negacyclic rotation of $\boldsymbol{f}_0$ by $\langle \vec{\alpha}, \vec{s} \rangle$ positions. The difference between the two bootstrapping procedures is that

- AP works by including in the evaluation key encryptions $E(\alpha \cdot s_i)$ for all possible values of α and then using α_i as a selector to pick one of them. This allows using arbitrary keys s_i with no impact on the running time, but also requires large evaluation keys due to the need to store multiple encryptions $E(\alpha \cdot s_i)$ for every secret key element s_i.[8]

[7]Under a typically different scheme $E(\cdot)$, used when generating the evaluation key.

[8]The method also offers storage memory trade-offs, decomposing a into a sequence of smaller "digits", but the same remarks apply.

- GINX on the other hand works by assuming $s_i \in \{0, 1\}$ is a single bit, and using $E(s_i)$ as a selector between the original ciphertext $\mathsf{RLWE}(\boldsymbol{f}_{i-1})$ and a modified one $\mathsf{RLWE}(\boldsymbol{f}_{i-1} \cdot X^{\alpha_i})$, using a homomorphic "MUX" gate. This only requires a single encryption $E(s_i)$ for each key element, but it is directly applicable only to binary secrets. Larger secrets can be handled in a number of ways, but not without a cost either in terms of key size or computation time. For example, [MP21] shows how to handle k-bit secrets by increasing both the evaluation key size and the bootstrapping running time, each by a factor k. A different tradeoff is given in [JP22], which incurs a smaller increase in running time (for small values of k) but at the cost of increasing the key size by an exponential factor $2^k - 1$. Both methods also result in higher bootstrapping noise.

In this paper, we present new techniques and optimizations to perform blind rotations using ring automorphisms and key switching. In its most basic form, the idea is the following: given $\mathsf{RLWE}(\boldsymbol{f}_{i-1}(X))$, one can first apply a ring autormorphsim[9] $\psi_{1/\alpha_i}(\cdot)$ where $\psi_a(\boldsymbol{h}) := \boldsymbol{h}(X^a)$. This gives an encryption of $\boldsymbol{f}_{i-1}(\alpha^{-1})$. Next, we homomorphically multiply the ciphertext by X^{s_i}, to get an encryption of $\boldsymbol{f}_{i-1}(\alpha^{-1}) \cdot X^{s_i}$. Finally, we apply the ring automorphism $\psi_{\alpha_i}(\cdot)$ to get an encryption of $\boldsymbol{f}_{i-1}(X) \cdot X^{\alpha_i s_i}$. After repeating this process n times, we obtain the encryption of $\mathsf{RLWE}(\boldsymbol{f}_0 \cdot X^{\langle \vec{\alpha}, \vec{s} \rangle})$.

The idea of using automorphisms is not new. For example, it was already used in a different context by Halevi and Shoup [HS18] to implement linear transformations and permutation networks in the (BGV-based) HElib HE library. Closely related to our use is the work of Bonnoron et al. (following a suggestion of Micciancio [BDF18, Footnote 6]), which uses automorphisms to reduce the key size of a variant of the FHEW cryptosystem. At a technical level, in [BDF18] the method is applied to the product of two cyclic polynomial rings, while we apply it to a single cyclotomic ring, as originally used by FHEW. As a result, the scheme of [BDF18] was not quite practical, resulting in poor concrete running times.

The basic algorithm based on automorphisms can be improved in a number of ways. For example, automorphisms from different steps can be composed together and replaced by a single automorphism. Other optimizations revolve around the technical difficulty that automorphisms ψ_α exist only for *odd* coefficients while bootstrapping requires multiplication by both even and odd values of α. Still, the resulting methods require a substantial number of "automorphism keys", to perform the required key switching after each application of ψ_a. We further improve the performance of the algorithm by introducing a new blind rotation strategy that reorders the secret key elements $s_1, \ldots, s_n$. Instead of iterating over the s_i (homomorphically multiplying by s_i), and the applying automorphism $\psi_{\alpha_i/\alpha_{i+1}}$ at each step, we iteratively apply a fixed automorphism ψ_g (where g generates a large subgroup $\mathbb{Z}_q^*$). This alone produces rotations $\boldsymbol{f}_0 \cdot X^{g^i}$ for all possible values of $g^i \in \mathbb{Z}_q^*$. Then, we intersperse the homomorphic multiplications by the secret key elements s_j when g^i is the correct value of α_j. The final result is $\mathsf{RLWE}(\boldsymbol{f}_0 \cdot X^{\langle \vec{\alpha}, \vec{s} \rangle})$ as desired, but using only a single automorphism key corresponding to the generator g. Notice how our proposed method is applicable to arbitrary keys, and its performance is independent of the range of the secret coordinates s_i.

In this intuitive explanation we omitted several important technicalities:

- The coefficients α_i are arbitrary integers in $\mathbb{Z}_q$, while automorphisms exist only for invertible

[9]The automorphism ψ_a alone maps an encryption under $\boldsymbol{z}(X)$ to an encryption under modified key $\boldsymbol{z}(X^a)$. Then, key-switching is used to turn this into an encryption under the original key $\boldsymbol{z}(X)$.

$\alpha \in \mathbb{Z}_q^*$.

- The multiplicative group $\mathbb{Z}_q^*$ is not cyclic, but factors as the product of two groups of size $q/4$ and 2
- In order to run over all of $\mathbb{Z}_q^*$, the automorphism g needs to be applied $O(q)$ times, but we would like the computation to take only $O(n)$ steps, independently of q.

The algorithms presented in the paper address all these difficulties and introduce further optimizations, which we analyze both in terms of worst-case and average-case complexity. As a result, our final algorithm achieves the best performance among all the known blind rotation techniques for FHEW-like cryptosystems both in terms of running time and key size.

Remark 1 *In practice, one can use Torus LWE or other similar structures over Ring LWE as demonstrated in [CGGI20]. To compare all bootstrapping methods observed in the same environment, we will use only Ring LWE in this paper following [MP21]. We note that it is straightforward to apply Torus LWE to each of the observed methods as it has shown in [CGGI20] for GINX.*

1.3 Applications to Threshold and Multi-key FHE

As described earlier, the linear properties of LWE allow to easily build threshold public-key encryption schemes: each party locally generates a key pair $p_i \approx a \cdot s_i$, and the public key $p = p_1 + \cdots + p_k$ can be set to the sum of the individual public keys. Things are more complex for threshold *FHE*. This is because, beside a public encryption key p, one needs to generate an *evaluation key*, which is essentially an encryption $E_p(s)$ of the secret key $s = s_1 + \cdots + s_k$ under the public key p. Naturally, this can be done using generic techniques from secure multiparty computation, with each party holding s_i as a local input, and common input p. However, this would not be quite practical. In fact, [AJLA$^+$12] gives specialized protocols to compute the evaluation key, but the method is specific to the BGV encryption scheme underlying their protocol. So, an interesting question is if a similar specialized evaluation key generation protocol can be designed for the threshold version of FHEW-like HE schemes. We observe that this is indeed possible, again using the linear homomorphic properties of lattice-based encryption. Specifically, after generating the global public key p, parties can encrypt their own secret shares $E_p(s_i)$ under it. Since all the shares are encrypted under a common public key, they can be added up, resulting in a HE $\sum_i E_p(s_i) = E_p(\sum_i s_i) = E_p(s)$ of the global secret key.[10]

Our blind rotation techniques also require the generation of switching keys to be used in conjunction with the ring automorphisms ψ_a. Again, a specialized distributed key generation algorithm can be built using the linearity of LWE encryption *and* the automorphisms. More in detail, in order to apply the automorphism ψ_a to a ciphertext, one needs to generate an encryption of the permuted secret key $E_p(\psi_a(s))$. Using the linearity of ψ_a, this can be achieved by having each party computing the encryption $E_p(\psi_a(s_i))$ of a permuted key share, and then combining these ciphertexts into $\sum_i E_p(\psi_a(s_i)) = E_p(\sum_i \psi_a(s_i)) = E_p(\psi_a(\sum_i s_i)) = E_p(\psi_a(s))$. The difference with standard (non-threshold) key generation, is that when the evaluation key is computed by a single party, the switching key $E_p(\psi_a(s))$ can be computed using a more efficient (and less noisy) private key version

[10]Calculation of $\sum_i E_p(s_i)$ proposed in this paper is done by the products of RGSW ciphertexts encrypting secret shares (see Section 6.)

of LWE encryption $E_s(\psi_a(s))$. Here, in order to distribute the computation among parties that only have shares s_i of the secret key, encryption is performed using the common public key p.

Another potential application of our techniques is *Multi-Key* HE. This is a generalization of HE where messages can be encrypted under independently generated public keys $p_1, \ldots, p_k$, and still allow to perform joint computations on them. Naturally, decrypting the final result requires knowledge of all relevant secret keys $s_1, \ldots, s_k$. So, this is similar to threshold encryption, but with the difference that $p_1, \ldots, p_k$ are not combined in advance into a single public key p, and the set of keys can be chosen dynamically. GSW-based (e.g., FHEW-like) multi-key HE schemes were proposed in a sequence of works [CM15, MW16, PS16, BP16, CCS19]. These schemes typically work by combining ciphertexts encrypted under different keys into a "multi-ciphertext", corresponding to the concatenation of the keys. Since the secret (decryption) key is also a concatenation, if the individual keys s_i are binary (as is the case for example in [CCS19]), their concatenation is also binary, and one can make direct use of the efficient GINX bootstrapping for binary keys. However, these concatenated "multi-ciphertexts" are much longer than simple RLWE encryption, and the cost of bootstrapping (compared to the single key setting) is even higher than GINX with large keys. (Specifically, it grows linearly with the number of parties, rather than logarithmically.) Recently, [ZZC$^+$21] have proposed a multi-key HE scheme with compact ciphertexts. Interestingly, this compact scheme combines the individual secret keys by taking their sum. So, it requires an efficient bootstrapping method with non-binary keys. Similar to the threshold encryption setting, our techniques can be applied to speed up bootstrapping while keeping a small evaluation key.

1.4 Other Important Related Works

Besides bootstrapping of FHEW/TFHE, blind rotation is a useful tool to evaluate arbitrary functions in HE. For example, the Cheon-Kim-Kim-Song (CKKS) scheme [CKKS17] is efficient in the evaluation of complex numbers, but it only supports addition and multiplication. Thus, the ReLU and comparison functions, which are important components of neural networks, are evaluated using blind rotation in [BGGJ20, LHH$^+$21, CJP21, LMP21] as they are not represented as polynomials in real numbers. Also, a generalized bootstrapping for all the RLWE-based HE schemes including CKKS, Brakerski-Gentry-Vaikuntanathan(BGV) [BGV14], and Brakerski/Fan-Vercauteren (BFV) [Bra12, FV12], was proposed using blind rotation in [KDE$^+$21].

1.5 Organization

The rest of the paper is organized as follows. The basic lattice-based HE and the previous blind rotation techniques are presented in Section 2. In Section 3, a new blind rotation algorithm and its variants are proposed. The theoretical analysis and comparison to prior works are given in Section 4 and the implementation results are given in Section 5. In Section 6, a threshold HE scheme based on our proposed blind rotation is described as a possible application. Finally, we conclude with remarks in Section 7.

2 Preliminaries

Let N be a power of two. We denote the $2N$-th cyclotomic ring by $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$ and its quotient ring by $\mathcal{R}_Q := \mathcal{R}/Q\mathcal{R}$. Ring elements in $\mathcal{R}$ are indicated in bold, e.g. $\boldsymbol{a} = \boldsymbol{a}(X)$. For two vectors $\vec{a}$ and $\vec{b}$, we denote their inner product by $\langle \vec{a}, \vec{b} \rangle$. We denote a vector of ones of length n by $\vec{\mathbf{1}}_n$. Function composition of $\boldsymbol{f}$ and $\boldsymbol{g}$ is denoted by $\boldsymbol{f} \circ \boldsymbol{g}$. All logarithms are base 2 unless otherwise indicated. We write the floor, ceiling and round functions as $\lfloor\cdot\rfloor$, $\lceil\cdot\rceil$ and $\lfloor\cdot\rceil$, respectively. We denote the infinity norm of $\boldsymbol{a} \in \mathcal{R}$ by $\|\boldsymbol{a}\|_\infty$, which is the largest absolute value among its coefficients. For $q \in \mathbb{Z}$ and $q > 1$, we identify the ring $\mathbb{Z}_q$ with $[-q/2, q/2)$ as the representative interval, and for $x \in \mathbb{Z}$ we denote the centered remainder of x modulo q by $[x]_q \in \mathbb{Z}_q$. We extend these notations to elements of $\mathcal{R}$ by applying them coefficient-wise. We use $\boldsymbol{a} \leftarrow \mathsf{S}$ to denote uniform sampling from the set S. We denote sampling according to a distribution χ by $\boldsymbol{a} \leftarrow \chi$.

2.1 Basic Lattice-based Encryption

For positive integers q and n, basic LWE encryption of $m \in \mathbb{Z}_q$ under the secret key $\vec{s} \leftarrow \chi_{\texttt{key}}$ is defined as

$$\mathsf{LWE}_{q,\vec{s}}(m) = (\vec{\alpha}, \beta) = (\vec{\alpha}, -\langle\vec{\alpha}, \vec{s}\rangle + e + m) \in \mathbb{Z}_q^{n+1},$$

where $\vec{\alpha} \leftarrow \mathbb{Z}_q^n$ and error $e \leftarrow \chi_{\texttt{err}}$. We occasionally drop subscripts q and $\vec{s}$ when they are obvious from the context.

For a positive integer Q and a power of two N, basic RLWE encryption of $\boldsymbol{m} \in \mathcal{R}$ under the secret key $\boldsymbol{z} \leftarrow \chi_{\texttt{key}}$ is defined as

$$\mathsf{RLWE}_{Q,\boldsymbol{z}}(\boldsymbol{m}) := (\boldsymbol{a}, -\boldsymbol{a} \cdot \boldsymbol{z} + \boldsymbol{e} + \boldsymbol{m}) \in \mathcal{R}_Q^2,$$

where $\boldsymbol{a} \leftarrow \mathcal{R}_Q$, and $e_i \leftarrow \chi_{\texttt{err}}$ for each coefficient e_i of $\boldsymbol{e}$ where $i \in [0, N-1]$. As with LWE, we will occasionally drop subscripts Q and $\boldsymbol{z}$.

We say that $(\boldsymbol{t}_0, \cdots, \boldsymbol{t}_{d_g-1})$ is a gadget decomposition of $\boldsymbol{t} \in \mathcal{R}_Q$ if $\boldsymbol{t} = \sum_{i=0}^{d_g-1} g_i \cdot \boldsymbol{t}_i$ where $\vec{g} = (g_0, \ldots, g_{d_g-1})$ is a gadget vector, and $\|\boldsymbol{t}_i\|_\infty < B_g$. We adapt the definitions of RLWE' and RGSW from [MP21]. For a gadget vector $\vec{g}$, we define $\mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{m})$ and $\mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m})$ as follows

$$\begin{aligned}
\mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{m}) &:= \big(\mathsf{RLWE}_{\boldsymbol{z}}(g_0 \cdot \boldsymbol{m}), \mathsf{RLWE}_{\boldsymbol{z}}(g_1 \cdot \boldsymbol{m}), \cdots, \mathsf{RLWE}_{\boldsymbol{z}}(g_{d_g-1} \cdot \boldsymbol{m})\big) \in \mathcal{R}_Q^{2d} \\
\mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m}) &:= \big(\mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{z} \cdot \boldsymbol{m}), \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{m})\big) \in \mathcal{R}_Q^{2\times 2d}.
\end{aligned}$$

The scalar multiplication between an element in $\mathcal{R}_Q$ and RLWE' ciphertext

$$\odot : \mathcal{R}_Q \times \mathsf{RLWE}' \to \mathsf{RLWE}$$

is defined as

$$\begin{aligned}
\boldsymbol{t} \odot \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{m}) &= \langle(\boldsymbol{t}_0, \cdots, \boldsymbol{t}_{d_g-1}), \big(\mathsf{RLWE}_{\boldsymbol{z}}(g_0 \cdot \boldsymbol{m}), \cdots, \mathsf{RLWE}_{\boldsymbol{z}}(g_{d_g-1} \cdot \boldsymbol{m})\big)\rangle \\
&= \sum_{i=0}^{d_g-1} \boldsymbol{t}_i \cdot \mathsf{RLWE}_{\boldsymbol{z}}(g_i \cdot \boldsymbol{m}) = \mathsf{RLWE}_{\boldsymbol{z}}\left(\sum_{i=0}^{d_g-1} g_i \cdot \boldsymbol{t}_i \cdot \boldsymbol{m}\right) \\
&= \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{t} \cdot \boldsymbol{m}) \in \mathcal{R}_Q^2,
\end{aligned}$$

For each error $\boldsymbol{e}_i$ in $\mathsf{RLWE}_{\boldsymbol{z}}(g_i \cdot \boldsymbol{m})$, the error after multiplication is equal to $\sum_{i=0}^{d_g-1} \boldsymbol{t}_i \cdot \boldsymbol{e}_i$ which is small if $\boldsymbol{t}_i$ and $\boldsymbol{e}_i$ are small.

The multiplication between RLWE and RGSW ciphertexts

$$\circledast : \mathsf{RLWE} \times \mathsf{RGSW} \to \mathsf{RLWE}$$

is defined as

$$\begin{aligned}\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{m}_1) \circledast \mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m}_2) &= (\boldsymbol{a}, \boldsymbol{b}) \circledast \left(\mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{z} \cdot \boldsymbol{m}_2), \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{m}_2)\right) \\ &= \boldsymbol{a} \odot \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{z} \cdot \boldsymbol{m}_2) + \boldsymbol{b} \odot \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{m}_2) \\ &= \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{a} \cdot \boldsymbol{z} \cdot \boldsymbol{m}_2) + \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{b} \cdot \boldsymbol{m}_2) \\ &= \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{m}_1 \cdot \boldsymbol{m}_2 + \boldsymbol{e}_1 \cdot \boldsymbol{m}_2) \in \mathcal{R}_Q^2.\end{aligned}$$

This result represents an RLWE encryption of the product $\boldsymbol{m}_1 \cdot \boldsymbol{m}_2$ with an additional error term $\boldsymbol{e}_1 \cdot \boldsymbol{m}_2$. In order to have $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{m}_1) \circledast \mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m}_2) \approx \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{m}_1 \cdot \boldsymbol{m}_2)$, it is necessary to make the error term $\boldsymbol{e}_1 \cdot \boldsymbol{m}_2$ small. This can be achieved by using monomials $\boldsymbol{m}_2 = \pm X^v$ as messages. The multiplication between $\mathsf{RLWE} \circledast \mathsf{RGSW}$ is naturally extended to $\mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m}_1) \circledast \mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m}_2) \approx \mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m}_1 \cdot \boldsymbol{m}_2)$.

Remark 2 *We note that for gadget vector $\vec{g} = (1, B_g, \ldots, B_g^{d_g-1})$, we can ignore $\boldsymbol{t}_0$ without a disadvantage and reduce runtime and key size. The error introduced by $\odot$ is $d_g N \frac{B_g^2}{12}\sigma^2$, where σ^2 is error variance of a fresh ciphertext. The error variance of $\sum_{i=1}^{d_g-1} \boldsymbol{t}_i \cdot \mathsf{RLWE}_{\boldsymbol{z}}(g_i \cdot \boldsymbol{m}) = \mathsf{RLWE}_{\boldsymbol{z}}\left(\sum_{i=1}^{d_g-1} g_i \cdot \boldsymbol{t}_i \cdot \boldsymbol{m}\right)$ is $(d_g-1)N\frac{B_g^2}{12}\sigma^2 + \mathsf{Var}(\boldsymbol{t}_0 \cdot \boldsymbol{m})$, where $\mathsf{Var}(\gamma)$ is the variance of random variable γ. Thus, it has less or equal error as long as $\mathsf{Var}(\boldsymbol{m}) \leq \sigma^2$, but saves one RLWE in RLWE′ and one NTT in $\odot$. This is similar to* approximate gadget decomposition *proposed in [CGGI20].*

2.1.1 Public-key Lattice-based Encryption

If an encryption of zero $\mathtt{pk}_{\boldsymbol{z}}^{\mathsf{RLWE}} = \mathsf{RLWE}_{\boldsymbol{z}}(0) = (\boldsymbol{a}, -\boldsymbol{a} \cdot \boldsymbol{z} + \boldsymbol{e})$ is given as a public key, then the public-key encryption can be done as

$$\mathtt{Enc}^{\mathsf{RLWE}}(\boldsymbol{m}; \mathtt{pk}_{\boldsymbol{z}}^{\mathsf{RLWE}}) := \boldsymbol{v} \cdot \mathtt{pk}_{\boldsymbol{z}}^{\mathsf{RLWE}} + (\boldsymbol{e}_0, \boldsymbol{m} + \boldsymbol{e}_1) = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{m}),$$

where $\boldsymbol{v} \leftarrow \chi_{\mathtt{key}}$, and $\boldsymbol{e}_0, \boldsymbol{e}_1 \leftarrow \chi_{\mathtt{err}}$.

We also can find encryption of $\boldsymbol{z} \cdot \boldsymbol{m}$ without the knowledge of $\boldsymbol{z}$ by slightly modifying the public key encryption (with the same amount of noise) as follows:

$$\mathtt{Enc}'^{\mathsf{RLWE}}(\boldsymbol{m}; \mathtt{pk}_{\boldsymbol{z}}^{\mathsf{RLWE}}) := \boldsymbol{v} \cdot \mathtt{pk}_{\boldsymbol{z}}^{\mathsf{RLWE}} + (\boldsymbol{m} + \boldsymbol{e}_0, \boldsymbol{e}_1) = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{z}\boldsymbol{m}).$$

Using $\mathtt{Enc}^{\mathsf{RLWE}}$ one can generate RLWE′ ciphertexts, and also can generate RGSW ciphertexts together with $\mathtt{Enc}'^{\mathsf{RLWE}}$ under the secret $\boldsymbol{z}$.

Key Switching in RLWE

The key switching operation converts a ciphertext $\mathsf{RLWE}_{\boldsymbol{z}_1}(\boldsymbol{m})$ encrypted under a secret key $\boldsymbol{z}_1$ to a ciphertext $\mathsf{RLWE}_{\boldsymbol{z}_2}(\boldsymbol{m})$ encrypted by a new secret key $\boldsymbol{z}_2$. There are different variants of the key switching technique and readers can refer to the literature (e.g., see [KPZ21]) for details. We focus on the BV key switching method [BV11]:

- $\mathtt{KSGen}(\boldsymbol{z}_1, \boldsymbol{z}_2)$: Outputs $\mathtt{swk} = \mathsf{RLWE}'_{\boldsymbol{z}_2}(\boldsymbol{z}_1)$.
- $\mathtt{KS}_{\boldsymbol{z}_1 \to \boldsymbol{z}_2}(\mathsf{RLWE}_{\boldsymbol{z}_1}(\boldsymbol{m}), \mathtt{swk})$: Given $\mathsf{RLWE}_{\boldsymbol{z}_1}(\boldsymbol{m}) = (\boldsymbol{a}, \boldsymbol{b})$, it outputs

$$\mathsf{RLWE}_{\boldsymbol{z}_2}(\boldsymbol{m}) = \boldsymbol{a} \odot \mathsf{RLWE}'_{\boldsymbol{z}_2}(\boldsymbol{z}_1) + (0, \boldsymbol{b}) \pmod{Q}.$$

$\mathsf{RLWE}'_{\boldsymbol{z}_2}(\boldsymbol{z}_1)$ generated by $\mathtt{KSGen}$ is a public switching key. The key switching error is equal to the error of $\mathcal{R} \odot \mathsf{RLWE}'$ multiplication.

$$\begin{tikzcd}
\boxed{(\beta, \vec{\alpha}) = \mathsf{LWE}_{q,\vec{s}}(q/4 \cdot m_0) + \mathsf{LWE}_{q,\vec{s}}(q/4 \cdot m_1)} \arrow[r, "\texttt{blind rotate}"] & \boxed{\mathsf{RLWE}_{Q,\boldsymbol{z}}\left(\boldsymbol{f} \cdot Y^{\beta + \langle \vec{\alpha}, \vec{s} \rangle}\right)} \arrow[r, "\texttt{LWE ext.}"] & \boxed{\mathsf{LWE}_{Q,\vec{z}}(Q/4 \cdot (m_0 \barwedge m_1))} \arrow[dll, "\texttt{mod switch}"] \\
\boxed{\mathsf{LWE}_{Q',\vec{z}}(Q'/4 \cdot (m_0 \barwedge m_1))} \arrow[r, "\texttt{key switch}"] & \boxed{\mathsf{LWE}_{Q',\vec{s}}(Q'/4 \cdot (m_0 \barwedge m_1))} \arrow[r, "\texttt{mod switch}"] & \boxed{\mathsf{LWE}_{q,\vec{s}}(q/4 \cdot (m_0 \barwedge m_1))}
\end{tikzcd}$$

Figure 1: NAND gate bootstrapping procedure of FHEW scheme [DM15, MP21].

Automorphisms in RLWE

In order to perform some operations in HE, we use the automorphisms of $\mathcal{R}$. There are N automorphisms $\psi_t : \mathcal{R} \to \mathcal{R}$ given by $\boldsymbol{a}(X) \mapsto \boldsymbol{a}(X^t)$ for $t \in \mathbb{Z}_{2N}^*$. We naturally extend ψ_t to $\mathcal{R}^2$ to apply the automorphism on a RLWE ciphertext. Automorphisms are applied using the following procedures which make use of a special set of switching keys $\mathtt{ak}_t = \mathsf{RLWE}_{\boldsymbol{z}(X)}(\boldsymbol{z}(X^t))$:

- $\mathtt{EvalAuto}_t\,(\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{m}), \mathtt{ak}_t)$: Given $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{m}(X)) = (\boldsymbol{a}(X), \boldsymbol{b}(X))$ and switching key $\mathtt{ak}_t$, apply ψ_t to $\boldsymbol{a}(X)$ and $\boldsymbol{b}(X)$ to obtain $(\boldsymbol{a}(X^t), \boldsymbol{b}(X^t))$, which is an RLWE encryption of $\boldsymbol{m}(X^t)$ under the secret key $\boldsymbol{z}(X^t)$. Then apply the key switching function $\mathtt{KS}_{\boldsymbol{z}(X^t) \to \boldsymbol{z}(X)}$ on the $\mathsf{RLWE}_{\boldsymbol{z}(X^t)}(\boldsymbol{m}(X^t))$ ciphertext, to produce the final output ciphertext $\mathsf{RLWE}_{\boldsymbol{z}(X)}(\boldsymbol{m}(X^t))$.

We note that ψ is a permutation on the coefficients of the elements of $\mathcal{R}$, which is easily calculated. ψ_t does not introduce additional error as an automorphism ψ_t is a norm-preserving map.

2.2 FHEW-like Bootstrapping

We briefly explain FHEW-like bootstrapping for NAND gates [DM15, MP21]. FHEW-like NAND gate bootstrapping starts with two $\mathsf{LWE}_{q,\vec{s}}$ ciphertexts with a small modulus q and adds them (HomNAND). After blind rotation and extraction procedures, we obtain an $\mathsf{LWE}_{Q,\vec{z}}$ encryption of the result with a higher ciphertext modulus Q. Using a sequence of modulus and key switchings we get back to an $\mathsf{LWE}_{q,\vec{s}}$ ciphertext. The bootstrapping procedure is shown in Figure 1. We focus on the blind rotation part and refer to [MP21] for more details on other parts of FHEW-like bootstrapping.

2.2.1 Blind Rotation

Blind rotation is an operation that multiplies a given ring element $\boldsymbol{f} \in \mathcal{R}_Q$ by a monomial Y^u, where the exponent $u = \beta + \langle \vec{\alpha}, \vec{s} \rangle \in \mathbb{Z}_q$ is given by an LWE ciphertext $(\vec{\alpha}, \beta) \in \mathbb{Z}_q^{n+1}$ encrypted under a secret key $\vec{s} \in \mathbb{Z}_q^n$. The output of the blind rotation is an RLWE encryption of $\boldsymbol{f} \cdot Y^u$, where q is small in practice ($q \approx 2^{10}$ in [DM15, CGGI20] and $q \approx 2^{12}$ in [KDE+21]). The operation is called "blind rotation" because it rotates the coefficients of $\boldsymbol{f}$ negacyclically, by an amount u which is provided in encrypted form. A formal definition is given below.

Definition 1 (Blind Rotation) *For $q|2N$, let $Y = X^{\frac{2N}{q}}$. A blind rotation is an algorithm which takes as input a ring element $\boldsymbol{f} \in \mathcal{R}_Q$, an $\mathsf{LWE}_{q,\vec{s}}$ ciphertext $(\vec{\alpha}, \beta) \in \mathbb{Z}_q^{n+1}$, and blind rotation keys $\boldsymbol{\mathtt{brk}}_{\boldsymbol{z},\vec{s}}$ corresponding to secrets $\boldsymbol{z}$ and $\vec{s}$, and outputs an RLWE ciphertext*

$$\mathsf{RLWE}_{Q,\boldsymbol{z}}\left(\boldsymbol{f} \cdot Y^{\beta + \langle \vec{\alpha}, \vec{s} \rangle}\right) \in \mathcal{R}_Q^2.$$

Two different blind rotation algorithms were proposed in [DM15, CGGI20]. Following [MP21], we refer to the two algorithms as “AP blind rotation” and “GINX blind rotation” respectively, as they are optimized ring versions of two bootstrapping procedures (for general LWE) originally proposed in [AP14] (AP) and [GINX16] (GINX). Both methods rely on the properties of RGSW ciphertexts described above.

AP Blind Rotation

In AP blind rotation [DM15, AP14], the blind rotation keys are generated for each element $s_i \in \mathbb{Z}_q$ of the secret $\vec{s}$ as

$$\mathtt{brk}^{AP} = \{\mathtt{brk}_{i,j,v} = \mathsf{RGSW}_{\boldsymbol{z}}(Y^{vB_r^j s_i})\}_{i,j,v}$$

for $i \in [0, n-1]$, $j \in \left[0, \log_{B_r}(q) - 1\right]$, and $v \in \mathbb{Z}_{B_r}$. In the algorithm, $\mathtt{acc}$ is initialized to the trivial encryption $\mathtt{acc} = \mathsf{RLWE}_{Q,\boldsymbol{z}}(\boldsymbol{f} \cdot Y^\beta) = (0, \boldsymbol{f} \cdot Y^\beta)$. Then, for each $i \in [0, n-1]$, α_i is decomposed in base B_r as $\alpha_i = \sum_{j=0}^{\log_{B_r}(q)-1} \alpha_{i,j} B_r^j$ and $\mathtt{acc}$ is updated sequentially for all $\alpha_{i,j}$ as

$$\mathtt{acc} \leftarrow \mathtt{acc} \circledast \mathsf{RGSW}_{\boldsymbol{z}}(Y^{\alpha_{i,j} B_r^j s_i}).$$

The full procedure of AP blind rotation is described in Algorithm 1.

Algorithm 1 Blind Rotation: AP [DM15, AP14]

```
1: procedure BlindRotateAP($\boldsymbol{f}, (\vec{\alpha}, \beta), \{\mathtt{brk}_{i,j,v} = \mathsf{RGSW}_{\boldsymbol{z}}(Y^{vB_r^j s_i})\}_{i,j,v}$)
2:     $\mathtt{acc} \leftarrow (0, \boldsymbol{f} \cdot Y^\beta)$
3:     for ($i = 0; i < n; i = i + 1$) do
4:         for ($j = 0; j < \log_{B_r}(q); j = j + 1$) do
5:             $\alpha_{i,j} = \lfloor \alpha_i / B_r^j \rfloor \pmod{B_r}$
6:             $\mathtt{acc} \leftarrow \mathtt{acc} \circledast \mathtt{brk}_{i,j\alpha_{i,j}}$
7: return $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^u)$
```

AP blind rotation supports all types of secret key distributions and provides a useful tradeoff between space and computational complexity based on the choice of the base $B_r \geq 2$. Greater B_r allows performing computations faster at the cost of storing more rotation keys, while smaller B_r reduces storage overhead but increases computational time.

GINX Blind Rotation

GINX blind rotation [CGGI20, GINX16] is more efficient than AP when the secret key $\vec{s}$ is set to a binary or ternary vector, but its performance degrades when using larger secret keys [MP21]. In the general case, each secret key element $s_i \in \mathbb{Z}_q$, $i \in [0, N-1]$, is expressed as subset-sum $s_i = \sum_{j=0}^{|U|-1} u_j \cdot s_{i,j}$ where $s_{i,j} \in \{0, 1\}$ and $U \subset \mathbb{Z}_q$ is an appropriately chosen subset of $\mathbb{Z}_q$. To express arbitrary elements of $\mathbb{Z}_q$ one can use $U = \{1, 2, 4, \ldots, 2^{k-1}\}$. But one can also use $U = \{1\}$ and $U = \{1, -1\}$ for binary and ternary secrets, respectively [MP21]. Using this notation for any fixed set U, the blind rotation key is generated as

$$\mathtt{brk}^{GINX} = \{\mathtt{brk}_{i,j} = \mathsf{RGSW}_{\boldsymbol{z}}(s_{i,j})\}$$

where $i = 0, \ldots, n-1$ and $j = 0, \ldots, |U|-1$. In the algorithm, $\mathtt{acc}$ is initiated to $\mathtt{acc} = \mathsf{RLWE}_{Q,\boldsymbol{z}}(\boldsymbol{f} \cdot Y^\beta) = (0, \boldsymbol{f} \cdot Y^\beta)$ and updated as

$$\mathtt{acc} \leftarrow \mathtt{acc} + (Y^{\alpha_i u_j} - 1) \cdot (\mathtt{acc} \circledast \mathsf{RGSW}_{\boldsymbol{z}}(s_{i,j})).$$

If $s_{i,j} = 0$, the second addendum is ignored since it gives an encryption of 0 and the value stored by the accumulator stays the same. If $s_{i,j} = 1$, then $\mathtt{acc} \circledast \mathsf{RGSW}_{\boldsymbol{z}}(1)$ is equal to $\mathtt{acc}$ and the accumulator is updated to $Y^{\alpha_i u_j} \cdot \mathtt{acc}$. Repeating this procedure for all $j \in [0, |U|-1]$ results in $Y^{\alpha_i s_i} \cdot \mathtt{acc}$. The full procedure for GINX blind rotation is described in Algorithm 2.

Algorithm 2 Blind Rotation: GINX [CGGI20, GINX16, MP21]

```
1: procedure BlindRotateGINX($\boldsymbol{f}, (\vec{\alpha}, \beta), \{\mathtt{brk}_{i,j} = \mathsf{RGSW}_{\boldsymbol{z}}(s_{i,j}) \mid s_i = \sum s_{i,j} u_j\}$)
2:     $\mathtt{acc} \leftarrow (0, \boldsymbol{f} \cdot Y^{\beta})$
3:     for $(i = 0; i < n; i = i + 1)$ do
4:         for $(j = 0; j < |U|; j = j + 1)$ do
5:             $\mathtt{acc} \leftarrow \mathtt{acc} + (Y^{\alpha_i u_j} - 1) \cdot (\mathtt{acc} \circledast \mathtt{brk}_{i,j})$
6: return $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{u})$
```

It is easy to see that for the small U this procedure may be efficient in both key size and running time. However, the running time and storage overhead grow significantly with larger secret key distributions. GINX blind rotation is more efficient than AP for secret keys $\vec{s}$ chosen from small distributions such as binary or ternary secret keys; but less efficient for general key size.

There is another optimization of GINX to remove the second loop in Algorithm 2 in such a way that it has about half of the computations and the same key size and error for ternary keys. However, it is only optimized for ternary keys [KDE+21, BIP+22] and cannot be efficiently extended to larger keys. Joye and Paillier proposed a variant of GINX in [JP22], which is a generalization of methods in [MP21] and [KDE+21], however, they suggested using binary and ternary secrets as they are the most efficient.

3 New Blind Rotation Techniques

In this section, we present new blind rotation algorithms which improve on previous methods [AP14, GINX16, DM15, CGGI17, CGGI20, MP21, JP22] in terms of running time, public key size, or both. Our algorithms update an accumulator ciphertext $\mathtt{acc}$ initialized to $\mathtt{acc} = (\boldsymbol{0}, \boldsymbol{f}') = \mathsf{RLWE}(\boldsymbol{f}')$, holding the encryption of a ring element $\boldsymbol{f}'$ related to $\boldsymbol{f}$, to be specified. The accumulator is updated through a sequence of $\mathsf{RLWE} \circledast \mathsf{RGSW}$ products, where RLWE holds the value of the accumulator $\mathtt{acc}$, and RGSW is an auxiliary ciphertext $\mathtt{brk}_i$ holding a secret key element s_i. Unlike previous techniques, our algorithms do not substitute multiplication in the exponent by series of additions (i.e. $\mathsf{RLWE} \circledast \mathsf{RGSW}$ products) but make use of ring automorphisms ψ_t and their associated switching keys $\mathtt{ak}_t$ instead.

For brevity, we first describe a core blind rotation algorithm for the case where $q = 2N$ and all α_i are odd since ψ_t is only defined for odd t, and then provide its variants and optimizations for other cases.

3.1 The Core Blind Rotation Algorithm

We recall that the goal of the algorithm is to rotate the accumulator by $Y^{\langle \vec{\alpha}, \vec{s} \rangle} = Y^{\sum_i \alpha_i s_i}$, where $\sum_i \alpha_i s_i$ is computed modulo $q = 2N$. For $N \geq 8$ the group $\mathbb{Z}_{2N}^*$ is isomorphic to $\mathbb{Z}_{N/2} \otimes \mathbb{Z}_2$ with generators $\{g, -1\}$ (e.g., $g = 5$) and every $t \in \mathbb{Z}_{2N}^*$ can be written as $\pm g^k$ where $k \in \mathbb{Z}_{N/2}$. Let $\alpha_i = \pm g^{k_i} \pmod{2N}$ for $i = 0, \ldots, n-1$. Let $I_\ell^+ = \left\{ i : \alpha_i = g^\ell \right\}$ and $I_\ell^- = \left\{ i : \alpha_i = -g^\ell \right\}$, for

$\ell \in [0, N/2-1]$. Using the fact that $g^{N/2} = 1 \pmod{2N}$ we have the following decomposition

$$\sum_i \alpha_i s_i = \left(\sum_{j\in I_0^+} s_j + \cdots + g\left(\sum_{j\in I_{N/2-1}^+} s_j - g\left(\sum_{j\in I_0^-} s_j + \cdots + g\left(\sum_{j\in I_{N/2-1}^-} s_j\right)\right)\right)\right) \pmod{2N}.$$

Denote $\texttt{brk}_j := \mathsf{RGSW}_{\boldsymbol{z}}\left(X^{s_j}\right)$. Given an initial ciphertext $\texttt{acc} = \mathsf{RLWE}^0_{\boldsymbol{z}}\left(\boldsymbol{f}'(X)\right)$, we first multiply it by $\texttt{brk}_j$ for all $j \in I^-_{N/2-1}$, then apply automorphism $\texttt{EvalAuto}_g$ to $\texttt{acc}$ and obtain

$$\texttt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}\left(\boldsymbol{f}'(X^g)\cdot X^{g\cdot\sum_{j\in I^-_{N/2-1}} s_j}\right).$$

Then we multiply the accumulator by $\texttt{brk}_j$ for $j \in I^-_{N/2-2}$ and again apply automorphism $\texttt{EvalAuto}_g$ to $\texttt{acc}$. This process is repeated for both I^-_ℓ and I^+_ℓ for all $\ell = N/2-1, ..., 0$. However, at the $(N/2)$th step (i.e., after multiplication by I^-_0) we apply the automorphism $\texttt{EvalAuto}_{-g}$ instead of $\texttt{EvalAuto}_g$, and (as an optimization) we skip the multiplication by the set I^+_0. The final result is

$$\texttt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}\left(\boldsymbol{f}'\left(X^{-g^{(N/2)-1}}\right)\cdot X^{\sum_i \alpha_i s_i}\right).$$

If we set $\boldsymbol{f}'(X) = \boldsymbol{f}\left(X^{-g}\right)\cdot X^{-g\beta}$, this equals $\texttt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}\left(\boldsymbol{f}(X)\cdot X^{\beta+\langle\vec{\alpha},\vec{s}\rangle}\right)$. During the computation, we use n keys $\texttt{brk}_i$ for $i \in [0, n-1]$ and two automorphism keys $\texttt{ak}_g$ and $\texttt{ak}_{-g}$. The algorithm performs two types of homomorphic operations: $\mathsf{RLWE} \circledast \mathsf{RGSW}$ multiplications and key switching for automorphisms. The number of $\mathsf{RLWE} \circledast \mathsf{RGSW}$ multiplications is n, and the number of automorphisms is $N-1$. We can reduce the number of automorphisms when some of the $I^\pm_\ell$ are empty because the automorphisms between them can be composed, and replaced by a single automorphism application. However, this requires storing a large number of automorphism keys $\texttt{ak}_{\pm g^u}$ for all possible values of u. Instead, for efficiency purposes, we store only a small number of keys $\{\texttt{ak}_{g^u}\}_{u\in[1,w]}$, for some parameter w which we call the *window size*. The full algorithm is provided in Algorithm 3. We will see in Section 4 that with a quite small window size we can achieve essentially the same improvement as when storing keys for all N possible automorphisms.

3.2 Dealing With Even α_i

We provide several solutions to overcome the issue with even α_i.

3.2.1 Memory Efficient Algorithm

One solution is to set $\omega_i = \alpha_i - 1$ if α_i is even and $\omega_i = \alpha_i$ if α_i is odd. Now we can apply the core blind rotation algorithm for the vector $\vec{\omega}$ and obtain $\mathsf{RLWE}\left(\boldsymbol{f}\cdot X^{\beta+\langle\vec{\omega},\vec{s}\rangle}\right)$. Then we repeatedly multiply $\texttt{brk}_i$ for each even α_i. This algorithm requires $n/2$ additional RGSW multiplications on average. If we store one additional key $\texttt{brk}_{\texttt{nsum}} := \mathsf{RGSW}(X^{-\sum_i s_i})$, and in case of the number of even α_i is greater than $n/2$, we initially multiply $\texttt{acc}$ by $\texttt{brk}_{\texttt{nsum}} := \mathsf{RGSW}(X^{-\sum_i s_i})$, and update $\alpha_i \leftarrow \alpha_i + 1$. This will make the number of odd α_i to be greater than half, mitigating the worst case. The full algorithm is provided in Algorithm 4.

3.2.2 Computation Efficient Algorithm

We can get rid of additional multiplications for even α_i in the previous solution by using auxiliary blind rotation keys $\texttt{brk}^*_i := \mathsf{RGSW}(X^{s_i+s_{i+1}})$, for $i \in [0, n-2]$. The idea is to find odd α'_i such that

Algorithm 3 Core Blind Rotation Sub Algorithm for odd α_i

```
1: procedure BlindRotateCore$\Big(\texttt{acc}, \vec{\alpha}, \{\texttt{brk}_i\}_{i\in[0,n-1]}, \{\texttt{ak}_{g^u}\}_{u\in[1,w]}, \texttt{ak}_{-g}\Big)$
2:     $v \leftarrow 0$
3:     for $(\ell = N/2-1; \ell > 0; \ell = \ell-1)$ do
4:         for $j \in I_\ell^-$ do
5:             $\texttt{acc} \leftarrow \texttt{acc} \circledast \texttt{brk}_j$
6:         $v \leftarrow v+1$
7:         if $(I_{\ell-1}^- \neq \emptyset$ or $v = w$ or $l = 1)$ then
8:             $\texttt{acc} \leftarrow \texttt{EvalAuto}_{g^v}(\texttt{acc}, \texttt{ak}_{g^v})$
9:             $v \leftarrow 0$
10:    for $j \in I_0^-$ do
11:        $\texttt{acc} \leftarrow \texttt{acc} \circledast \texttt{brk}_j$
12:    $\texttt{acc} \leftarrow \texttt{EvalAuto}_{-g}(\texttt{acc}, \texttt{ak}_{-g})$
13:    for $(\ell = N/2-1; \ell > 0; \ell = \ell-1)$ do
14:        for $j \in I_\ell^+$ do
15:            $\texttt{acc} \leftarrow \texttt{acc} \circledast \texttt{brk}_j$
16:        $v \leftarrow v+1$
17:        if $(I_{\ell-1}^+ \neq \emptyset$ or $v = w$ or $l = 1)$ then
18:            $\texttt{acc} \leftarrow \texttt{EvalAuto}_{g^v}(\texttt{acc}, \texttt{ak}_{g^v})$
19:            $v \leftarrow 0$
20:    for $j \in I_0^+$ do
21:        $\texttt{acc} \leftarrow \texttt{acc} \circledast \texttt{brk}_j$
22: return acc
```

Algorithm 4 Memory Efficient Blind Rotation Algorithm, $q = 2N$

```
1: procedure BlindRotateME$\Big(\boldsymbol{f}, \vec{\alpha}, \beta, \{\texttt{brk}_i\}_{i\in[0,n-1]}, \texttt{brk}_{\texttt{nsum}}, \{\texttt{ak}_{g^u}\}_{u\in[1,w]}, \texttt{ak}_{-g}\Big)$
2:     $\texttt{acc} \leftarrow (\mathbf{0}, \boldsymbol{f}\left(X^{-g}\right) \cdot X^{-g\beta})$
3:     if number of even $\alpha_i$ is $> n/2$ then
4:         $\texttt{acc} \leftarrow \texttt{acc} \circledast \texttt{brk}_{\texttt{nsum}}$
5:         $\vec{\alpha} \leftarrow \vec{\alpha} + \vec{\mathbf{1}}_n \pmod{2N}$
6:     for $(i = 0; i < n; i = i+1)$ do
7:         if $\alpha_i$ is even then
8:             $\omega_i \leftarrow \alpha_i - 1 \pmod{2N}$
9:         else
10:            $\omega_i \leftarrow \alpha_i$
11:    $\texttt{acc} \leftarrow \texttt{BlindRotateCore}(\texttt{acc}, \vec{\omega}, \{\texttt{brk}_i\}, \{\texttt{ak}_{g^u}\}, \texttt{ak}_{-g})$
12:    for $(i = 0; i < n; i = i+1)$ do
13:        if $\alpha_i$ is even then
14:            $\texttt{acc} \leftarrow \texttt{acc} \circledast \texttt{brk}_i$
15: return $\texttt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f}(X) \cdot X^{\beta + \langle\vec{\alpha},\vec{s}\rangle})$
```

$\sum_i \alpha_i s_i = \sum_i \alpha'_i s'_i$, where s'_i is either equal to s_i or to $s_i + s_{i+1}$. First, we assume that α_0 is odd and set $\alpha'_0 = \alpha_0$, otherwise, we initially multiply the accumulator by $\mathtt{brk_{nsum}}$ and update $\alpha_i \leftarrow \alpha_i + 1$. Then at each step i, assuming (by induction) that α'_i is odd, we consider two cases, depending on the parity of α_{i+1}. If α_{i+1} is odd, we set $s'_i = s_i$ and $\alpha'_{i+1} = \alpha_{i+1}$. Otherwise, we set $s'_i = s_i + s_{i+1}$ and balance this by setting $\alpha'_{i+1} = \alpha_{i+1} - \alpha_i$. In either case, the value of α'_{i+1} is odd, preserving the inductive hypothesis, and we may move to the next iteration. For the last iteration we always set $s'_{n-1} = s_{n-1}$. Note that during the process we do not need to know the values s_i, we only have the information of whether $s'_i = s_i$ or $s'_i = s_i + s_{i+1}$. The full algorithm is provided in Algorithm 5.

Algorithm 5 Computation Efficient Blind Rotation Algorithm, $q = 2N$

```
1: procedure BlindRotateCE$\Big(\boldsymbol{f}, (\vec{\alpha}, \beta), \{\mathtt{brk}_i\}_{i\in[0,n-1]}, \{\mathtt{brk}^*_i\}_{i\in[0,n-2]}, \mathtt{brk_{nsum}}, \{\mathtt{ak}_{g^u}\}_{u\in[1,w]}, \mathtt{ak}_{-g}\Big)$
2:     $\mathtt{acc} \leftarrow (\mathbf{0}, \boldsymbol{f}\left(X^{-g}\right) \cdot X^{-g\beta})$
3:     if $\alpha_0$ is even then
4:         $\mathtt{acc} \leftarrow \mathtt{acc} \circledast \mathtt{brk_{nsum}}$
5:         $\vec{\alpha} \leftarrow \vec{\alpha} + \vec{\mathbf{1}}_n \pmod{2N}$
6:     Find odd $\alpha'_i : \sum_i \alpha_i s_i = \sum_i \alpha'_i s'_i$
7:     for $(i = 0; i < n; i = i+1)$ do
8:         if $s'_i = s_i$ then
9:             $\mathtt{brk}'_i \leftarrow \mathtt{brk}_i$
10:        else
11:            $\mathtt{brk}'_i \leftarrow \mathtt{brk}^*_i$
12:    $\mathtt{acc} \leftarrow \mathtt{BlindRotateCore}(\mathtt{acc}, \vec{\alpha}', \{\mathtt{brk}'_i\}, \{\mathtt{ak}_{g^u}\}, \mathtt{ak}_{-g})$
13: return $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f}(X) \cdot X^{\beta + \langle\vec{\alpha}, \vec{s}\rangle})$
```

3.2.3 Case $q = N$

In FHEW-like cryptosystems [DM15, MP21, CGGI20], commonly the blind rotation input LWE ciphertext $(\vec{\alpha}, \beta)$ has a modulus $q < 2N$. The use of $q < 2N$ helps decrease the key size of AP-style bootstrapping. The size of q affects the decryption failure of LWE ciphertexts. However in practice, in the most interesting case, we can achieve $q = N$ with a negligible probability of decryption failure.

For our case we raise the modulus from N to $2N$, by multiplying the ciphertext $(\vec{\alpha}, \beta)$ by factor 2, resulting $(2\vec{\alpha}, 2\beta)$ with all even $2\alpha_i$. We initially multiply $\mathtt{acc}$ by $\mathtt{brk_{nsum}}$ to make all $2\alpha_i + 1$ to be odd. The full algorithm is provided in Algorithm 6.

Algorithm 6 Blind Rotation Algorithm, $q = N$

```
1: procedure BlindRotateOptim$\Big(\boldsymbol{f}, \vec{\alpha}, \beta, \{\mathtt{brk}_i\}_{i\in[0,n-1]}, \mathtt{brk_{nsum}}, \{\mathtt{ak}_{g^u}\}_{u\in[1,w]}, \mathtt{ak}_{-g}\Big)$
2:     $\mathtt{acc} \leftarrow (\mathbf{0}, \boldsymbol{f}\left(X^{-g}\right) \cdot X^{-2g\beta})$
3:     $\mathtt{acc} \leftarrow \mathtt{acc} \circledast \mathtt{brk_{nsum}}$
4:     $\vec{\alpha}' \leftarrow 2\vec{\alpha} + \vec{\mathbf{1}}_n \pmod{2N}$
5:     $\mathtt{acc} \leftarrow \mathtt{BlindRotateCore}(\mathtt{acc}, \vec{\alpha}', \{\mathtt{brk}_i\}, \{\mathtt{ak}_{g^u}\}, \mathtt{ak}_{-g})$
6: return $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f}(X) \cdot X^{2(\beta + \langle\vec{\alpha}, \vec{s}\rangle)})$
```

$$\begin{tikzcd}
\boxed{\mathsf{LWE}_{Q,\vec{z}}(Q/4 \cdot m_0) + \mathsf{LWE}_{Q,\vec{z}}(Q/4 \cdot m_1)} \arrow[r, "\texttt{mod switch}"] & \boxed{\mathsf{LWE}_{Q',\vec{z}}(Q'/4 \cdot m)} \arrow[r, "\texttt{key switch}"] & \boxed{\mathsf{LWE}_{Q',\vec{s}}(Q'/4 \cdot m)} \arrow[dll, "\texttt{mod switch}"] \\
\boxed{(\beta, \vec{\alpha}) = \mathsf{LWE}_{q,\vec{s}}(q/4 \cdot m)} \arrow[r, "\texttt{blind rotate}"] & \boxed{\mathsf{RLWE}_{Q,\boldsymbol{z}}\left(\boldsymbol{f} \cdot Y^{\beta + \langle \vec{\alpha}, \vec{s} \rangle}\right)} \arrow[r, "\texttt{LWE ext}"] & \boxed{\mathsf{LWE}_{Q,\vec{z}}(Q/4 \cdot (m_0 \barwedge m_1))}
\end{tikzcd}$$

Figure 2: NAND gate bootstrapping procedure of FHEW scheme. We start from $\mathsf{LWE}_{Q,\vec{z}}$ and switch to $\mathsf{LWE}_{q,\vec{s}}$ before blind rotation. We refer [MP21] for other gates.W

3.3 Improved FHEW Scheme and Removal of $\texttt{brk}_{\texttt{nsum}}$

As was mentioned in [DM15] we can reduce the noise and number of key switching operations in FHEW-like bootstrapping, by swapping some operations in the procedure in Figure 1. We start with a ciphertext with a higher modulus Q rather than q and do modulus switching to q right before the blind rotation. (See Figure 2.)

Here we propose a trick which we call *round-to-odd* to get all-odd LWE ciphertext during modulus reduction so that $\texttt{brk}_{\texttt{nsum}}$ in Algorithm 6 becomes unnecessary. Thus, the round-to-odd gives advantages in runtime, key size, and noise growth regarding multiplication of $\texttt{brk}_{\texttt{nsum}}$. For ciphertext $(\vec{\alpha}', \beta') = \mathsf{LWE}_{Q'}(Q'/4 \cdot m)$, the modulus reduction is defined as

$$\left(\vec{\alpha} = \left\lfloor \frac{q}{Q'} \cdot \vec{\alpha}' \right\rceil, \beta = \left\lfloor \frac{q}{Q'} \cdot \beta' \right\rceil\right) = \mathsf{LWE}_q(q/4 \cdot m).$$

We modify the rounding operation to round-to-odd, $\lfloor x \rceil_{\texttt{odd}}$, which returns the nearest odd integer for the given input x. In addition, if x is closer to zero than any other odd number (i.e., 1), it returns zero. Then the new modulus reduction is defined as

$$\left(\vec{\alpha} = \left\lfloor \frac{2N}{Q'} \cdot \vec{\alpha}' \right\rceil_{\texttt{odd}}, \beta = \left\lfloor \frac{2N}{Q'} \cdot \beta' \right\rceil_{\texttt{odd}}\right) = \mathsf{LWE}_{2N}(q/4 \cdot m),$$

which gives an LWE ciphertext of modulus $2N$ with all-odd coefficients. We note that the modulus reduction error by round-to-odd is equivalent to modulus switching to N. The blind rotation algorithm for the round-to-odd trick case is provided in Algorithm 7

Algorithm 7 Blind Rotation Algorithm with Round-to-odd Input, $q = 2N$

```
1: procedure BlindRotateRoundToOdd$\left(\boldsymbol{f}, \vec{\alpha}, \beta, \{\texttt{brk}_i\}_{i \in [0,n-1]}, \{\texttt{ak}_{g^u}\}_{u \in [1,w]}, \texttt{ak}_{-g}\right)$    ▷ $\vec{\alpha}, \beta$ are all odd
2:     $\texttt{acc} \leftarrow (\mathbf{0}, \boldsymbol{f}(X^{-g}) \cdot X^{-g\beta})$
3:     $\texttt{acc} \leftarrow \texttt{BlindRotateCore}(\texttt{acc}, \vec{\alpha}, \{\texttt{brk}_i\}, \{\texttt{ak}_{g^u}\}, \texttt{ak}_{-g})$
4: return $\texttt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f}(X) \cdot X^{\beta + \langle \vec{\alpha}, \vec{s} \rangle})$
```

4 Analysis

In this section, we analyze our new blind rotation technique and compare it to the prior art. We analyze blind rotation separately from the full FHEW scheme since blind rotation is a useful tool in a number of other applications, e.g., the homomorphic evaluation of non-polynomial functions [LHH+21, LMP21] and CKKS/BGV/BFV bootstrapping [KDE+21].

4.1 Analysis of the Number of Automorphisms

We focus on the number of automorphisms for Algorithm 3. First notice that the number of non-empty $I_\ell^\pm$ is always at most $\min(N, n)$ just because there are a total of N sets, and their union has size n, i.e., the total number of terms $\alpha_i s_i$. Moreover, it can be less than n if some of the s_i have the same coefficient α_i. We evaluate the average number of non-empty $I_\ell^\pm$ under the standard assumption that the LWE coefficients α_i are random and independent.[11] Assume without loss of generality that all α_i are odd, as enforced by our algorithms. Each fixed set $I_\ell^\pm$ is empty if all α_i do not belong to it. Since the α_i are uniform and independent, this happens with probability $(1-1/N)^n \approx e^{-n/N}$. Therefore $I_\ell^\pm$ is non-empty with probability $1-(1-1/N)^n \approx 1-e^{-n/N}$, and, by linearity of expectation, the expected number of nonempty sets is $N(1-(1-1/N)^n) \approx N(1-e^{-n/N})$.

Counting the number of non-empty sets $I_\ell^\pm$ is useful to estimate the number of automorphism applications performed by our algorithm because the automorphisms between non-empty sets are composed and replaced by a small number of automorphisms with keys in $\{\mathtt{ak}_{g^u}\}_{u\in[1,w]}$ for a given window size w. Let k be the number of non-empty sets $I_\ell^\pm$, be it either $\min(N, n)$ in the worst case, or $k = N(1 - e^{-n/N})$ on average. Let $v_1, \ldots, v_k$ be the exponents of the k automorphisms g^{v_i} that need to be applied after each non-empty set. Write each exponent as $v_i = v_i' + w \cdot v_i''$ where $v_i' = v_i \bmod w \in \{1, \ldots, w - 1\}$, and $v_i'' = \lfloor v_i'/w \rfloor$. (In case v_i is a multiple of w, the v_i' part can be omitted altogether.) In Algorithm 3, the v_i applications of the basic automorphism g (following multiplication by the ith set $I_\ell^\pm$) are replaced by one application of automorphism $g^{v_i'}$ and v_i'' applications of automorphism g^w. So, the number of automorphism applications of type $g^{v_i'}$ is κ, for some $\kappa \le k$.[12] In order to bound the number of applications of automorphism g^w, we use the fact that the sum $\sum_i v_i$ is bounded by N. Therefore, $\sum_i v_i''$ is at most $\frac{N-\kappa}{w}$. In summary, by storing w automorphism keys $\{\mathtt{ak}_{g^u}\}_{u\in[1,w]}$, we can reduce the number of automorphism applications to $\kappa + \frac{N-\kappa}{w} = (1-1/w)\kappa + (1/w)N \le (1-1/w)k + (1/w)N$. We always have $n \le N$, and in the worst case, we have $k \le n$. So the total number of automorphism applications is always bounded by $(1-1/w)n + (1/w)N$. On average, using $k \approx N(1-e^{-n/N})$, the expected number of automorphism applications reduces to $N(1 - (1 - 1/w) \cdot e^{-n/N})$.

4.2 Complexity, Key Size, and Error Analysis

The comparison of computational complexity, key size, and error are given in Table 1. In order to facilitate the comparison of all blind rotation algorithms, we measure their time complexity in terms of the number of $\mathcal{R} \odot \mathsf{RLWE}'$ products they perform, as the cost of these operations dominates the total running time. Each $\mathsf{RLWE} \circledast \mathsf{RGSW}$ product requires two $\odot$ multiplications, while key switching is performed with a single $\odot$ multiplication. So, both $\circledast$ products and key switching operations are easily expressed in terms of $\odot$ products. We note that the operation $\odot$ can be considered as an abstraction of a basic operation for FHEW and its torus variant TFHE [CGGI20]. Another common measure of complexity used in previous works on FHEW-like HE is the number of NTT/FFT performed by the algorithms. We note that one can easily convert the number of $\odot$ products to the number of NTT as each $\odot$ requires precisely $(d_g + 1)$ NTT operations, where

[11]This is certainly true for freshly encrypted messages, as the α_i are chosen uniformly at random by the encryption algorithm. But it is reasonable to expect this to be true even when the ciphertext is the result of a homomorphic computation.

[12]κ will be less than k if some of the v_i' are 0.

d_g is the number of elements of a gadget vector. We note that $\odot$ requires d_g NTT operations if approximate gadget decomposition is used.

Similarly, we compare the memory requirement of all blind rotation algorithms using the total number of RLWE' ciphertexts required by the blind rotation key. The blind rotation keys for all methods consist of several RGSW and RLWE' ciphertexts. In turn, each RGSW is composed of two RLWE' ciphertexts. For the sake of brevity, "blind rotation key size" refers to the size of both $\mathtt{brk}$ and $\mathtt{ak}$ in this section. This can be translated into a traditional "bit size" simply noting that each RLWE' ciphertext requires roughly $2d_gN\log Q$-bit of space (or $2(d_g-1)N\log Q$-bit with approximate gadget decomposition.)

We also note that in our analysis and implementation we use approximate gadget decomposition. Approximate gadget decomposition does not introduce additional error but reduces runtime and key size for all analyzed blind rotation techniques. One can find the counterparts for exact gadget decomposition by simply substituting d_g-1 with d_g in the equations.

We use an approach from [DM15, MP21] to estimate the variance $\sigma^2_{\mathtt{acc}}$ from the blind rotation procedure. The total error for algorithms using blind rotation such as FHEW/TFHE bootstrapping [DM15, MP21, CGGI20] and amortized FHEW bootstrapping [MS18], can be easily estimated using this value. The error variance introduced by a single $\odot$ operation is equal to $d_gN\frac{B_g^2}{12}\sigma^2$, where B_g and d_g are parameters for gadget decomposition used in $\odot$ multiplication. For the sake of brevity we denote $\sigma^2_{\odot} := d_gN\frac{B_g^2}{12}\sigma^2$.

In AP and our algorithms, each $\circledast$ is performed by RGSW encrypting the monomial, and thus introduces an additive error with variance $2\cdot\sigma^2_{\odot}$. The automorphism operation due to key switching introduces an additive error with variance $\sigma^2_{\odot}$. Thus the variance $\sigma^2_{\mathtt{acc}}$ can be estimated as $\sigma^2_{\odot}$ multiplied by the number of $\odot$ operations. In the GINX and GINX* variants, due to the preprocessing of RGSW ciphertexts before $\circledast$ multiplications, each $\circledast$ introduces an additive error with variance $4\cdot\sigma^2_{\odot}$ and $8\cdot\sigma^2_{\odot}$, respectively.

We note that the parameters for the FHEW scheme in Section 5 are selected following this theoretical analysis. However, the fact that one technique has a smaller complexity expression than another in this theoretical analysis does not necessarily mean that it will show a better runtime in practice, because of the use of different parameter sets required to achieve a target security level. For example, binary GINX has the smallest expression representing the abstract key size and runtime in this analysis. But in practice, our new blind rotation algorithm outperforms binary GINX because of the following two reasons. First, our blind rotation has less noise growth compared to binary GINX, allowing a smaller parameter set to be used. Second, we can achieve the same security level with a smaller n by using Gaussian secrets at no cost in performance.

5 Implementation

In this section, we present the implementation results of our new blind rotation algorithm as applied to FHEW bootstrapping. For our implementation, we use Algorithm 6 optimized by reducing the number of automorphisms, which gives the best performance. We compare it to the AP and GINX blind rotation techniques. According to the theoretical analysis presented in the previous section, similar results will be achieved for TFHE [CGGI20] by using floating-point operations and DFTs instead of operations over finite rings and NTTs, respectively for each discussed blind rotation technique.

Table 1: Complexity, key size, and error variance of each blind rotation technique. Key size (# keys) is the number of RLWE' ciphertexts, and computational complexity (# mult) is the number of $\mathcal{R} \odot \mathsf{RLWE}'$. The parameter w is a small integer, typically a small constant independent of n. The parameter $|U|$ depends on the secret key size and can be as large as $\log n$ for gaussian secrets following the error distribution.

Method	# keys (in RLWE')	# mult (in $\odot$)	$\sigma^2_{\mathtt{acc}}/\sigma^2_{\odot}$
AP [AP14, DM15]	$2d_r(B_r-1)n$	$2d_r\left(1-\frac{1}{B_r}\right)n$	$2d_r\left(1-\frac{1}{B_r}\right)n$
GINX [GINX16, CGGI20, MP21]	$2\|U\|n$	$2\|U\|n$	$4\|U\|n$
GINX* [KDE+21, BIP+22]	$4n$	$2n$	$8n$
Ours (Alg. 4)	$2n+w+3$	$3n+\frac{w-1}{w}\kappa+\frac{N}{w}$	$3n+\frac{w-1}{w}\kappa+\frac{N}{w}$
Ours (Alg. 5)	$4n+w+1$	$2n+\frac{w-1}{w}\kappa+\frac{N}{w}+2$	$2n+\frac{w-1}{w}\kappa+\frac{N}{w}+2$
Ours (Alg. 6)	$2n+w+3$	$2n+\frac{w-1}{w}\kappa+\frac{N}{w}+2$	$2n+\frac{w-1}{w}\kappa+\frac{N}{w}+2$
Ours (Alg. 7)	$2n+w+1$	$2n+\frac{w-1}{w}\kappa+\frac{N}{w}$	$2n+\frac{w-1}{w}\kappa+\frac{N}{w}$

Table 2: Optimized parameter sets for FHEW schemes. Error variance is 3.2 and for TFHE, we put error variance instead of q and Q as it is defined over Torus

Parameter set	key	n	q	N	Q	$Q_{\mathtt{ks}}$	d_g	$d_{\mathtt{ks}}$	$\lambda_{\min}$
128_Ours/AP	$\sigma = 3.2$	458	1024	1024	2^{28}	2^{14}	3	2	128.2
128_tGINX	ternary	531	2048	1024	2^{26}	2^{14}	4	2	128.5
128_bGINX	binary	571	2048	1024	2^{25}	2^{14}	4	2	128.1
STD128_OPT [MP21]	ternary	502	1024	1024	2^{27}	2^{14}	4	2	121.0
TFHE [TFH]	binary	630	$\sigma = 2^{-15}$	1024	$\sigma = 2^{-25}$	–	3	2	115.11

5.1 Parameter Sets

The full procedure for FHEW bootstrapping is presented in Figure 2. Using the unique characteristics of each blind rotation technique and the choice of secret key distribution, in Table 2 we provide optimized parameter sets for FHEW schemes with AP, GINX, and our new technique. Following to [DM15], we choose the best parameters to have the smallest key size and runtime while keeping the gate bootstrapping (NAND) failure probability below 2^{-32}. According to these criteria, we propose new 128-bit secure parameter sets 128_Ours/AP, 128_tGINX, and 128_bGINX for Ours/AP with Gaussian secrets, GINX* with ternary secrets, and GINX with binary secrets, respectively. For comparison purposes, Table 2 also provides optimized parameters for AP and GINX from previous works which have smaller security considering the latest cryptoanalysis. The security is estimated using the lattice estimator [APS15] (commit 09e235).

Let $\sigma^2_{\mathtt{ms1}}$, $\sigma^2_{\mathtt{ks}}$, and $\sigma^2_{\mathtt{ms2}}$ denote the error variances introduced by modulus switching from Q to $Q' < Q$, key switching from $\vec{z}$ to $\vec{s}$, and modulus switching from Q' to q, respectively. $\mathsf{LWE}_{q,\vec{s}}(q/4 \cdot m)$ has the greatest noise, whose variance is

$$\varsigma^2 = \frac{q^2}{Q^2} \cdot 2\sigma^2_{\mathtt{acc}} + \frac{q^2}{Q'^2}\left(\sigma^2_{\mathtt{ks}} + \sigma^2_{\mathtt{ms1}}\right) + \sigma^2_{\mathtt{ms2}}.$$

Table 3: Bootstrapping failure probability of each blind rotation method. The failure probability of Alg. 7 is estimated for the worst case, i.e., the number of automorphism is $(1-1/w)n+(1/w)N$.

Parameter set	Alg. 7	AP	GINX*	GINX-binary
128_Ours/AP	$2^{-43.51}$	$2^{-40.22}$	x	x
128_tGINX	$2^{-60.03}$	$2^{-54.24}$	$2^{-35.79}$	x
128_bGINX	$2^{-40.68}$	$2^{-37.26}$	x	$2^{-37.26}$
STD128_OPT [MP21]	$2^{-57.81}$	$2^{-55.91}$	$2^{-53.65}$	x
TFHE [TFH]	$2^{-37.77}$	$2^{-30.57}$	x	$2^{-30.57}$

Table 4: Timing results (average of 400), blind rotation key size, and failure probability for FHEW bootstrapping (NAND gate).

Parameter set	Method	Runtime [ms]	Key size [MB]	Fail. prob.
128_Ours/AP	Alg. 7	**80.1**	**12.67**	$2^{-43.51}$
128_Ours/AP	AP	127.8	776.45	$2^{-40.22}$
128_tGINX	GINX*	89.7	40.45	$2^{-35.79}$
128_bGINX	GINX	84.1	20.91	$2^{-37.26}$

Similar to [DM15] we estimate

$$\sigma^2_{\mathtt{ms1}} = \frac{\|\vec{z}\|^2+1}{12}, \sigma^2_{\mathtt{ks}} = \sigma^2 N d_{\mathtt{ks}}, \sigma^2_{\mathtt{ms2}} = \frac{\|\vec{s}\|^2+1}{12}.$$

We assume $\|\vec{z}\| \le \sqrt{N/2}$ and $\|\vec{s}\| \le \sqrt{n/2}$ for binary or ternary secrets [DM15], and $\|\vec{z}\| = \sqrt{N\sigma^2}$ and $\|\vec{s}\| = \sqrt{n\sigma^2}$ for Gaussian secrets. The decryption fails when the noise of $\mathsf{LWE}_{q,\vec{s}}(q/4 \cdot m)$ exceeds $q/8$, and thus the decryption failure probability per NAND is given by $1 - \mathrm{erf}(\frac{q/8}{2\varsigma})$.

Table 3 provides the estimated bootstrapping failure probability $1 - \mathrm{erf}(\frac{q/8}{2\varsigma})$. It shows that among all the existing methods, the proposed blind rotation has the least error. As our blind rotation and AP take advantage of smaller q, we replaced $q = 1024$ for our blind rotation and AP in this table. In this estimate, we set $w = 10$.

5.2 Runtime Results

In order to provide a fair comparison of bootstrapping algorithms, we have implemented all of them using identical libraries and computing environments. The evaluation environment is PALISADE v.1.11.5 on Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz, running Ubuntu 20.04.3 LTS. We compiled with `clang 12` and the following CMake flags: NATIVE_SIZE=32, WITH_OPENMP=OFF, WITH_NATIVEOPT=ON.

Table 4 shows runtime results and blind rotation key size for NAND gate evaluation of FHEW. We provide experimental results for different parameter sets for binary, ternary, and Gaussian secret key distributions. This table demonstrates that the proposed algorithm with the parameter set 128_Ours/AP has the best performance. In this experiment, we set the window size w to 10.

The impact of different window sizes is demonstrated in Figure 5.2 where runtime results for NAND gate evaluation of FHEW are presented depending on the window size. We can see that

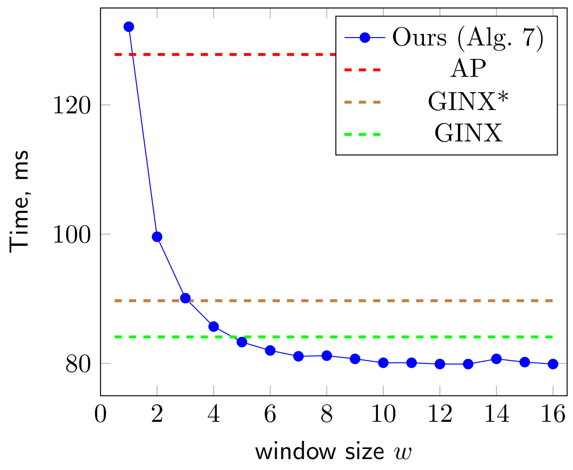

Figure 3: Bootstrapping performance results of Alg. 7 method for different window sizes

with a window size of 10 and greater, the running time of the proposed blind rotation technique is approximately the same. This is consistent with our complexity analysis in Section 3.

6 Applications to Threshold Homomorphic Encryption

The proposed blind rotation technique supports arbitrary secret key distribution and has reasonable complexity and small key size with the smallest blind rotation error among all known blind rotation techniques. In this section, we outline a threshold HE scheme which takes advantage of the proposed blind rotation technique. The simple structure of our blind rotation keys gives us an instinctive design of FHEW-like threshold HE with the approach proposed in [AJLA+12].

Following the basic concept described in Section 1.3, to enable threshold HE using the FHEW scheme, we define the algorithms for distributed evaluation key generation. Each participant j has the secret keys $\vec{s}_j$ for LWE encryption and $\boldsymbol{z}_j$ for RLWE encryption, where $j \in J$ and J denotes the set of participants with $|J| = k$. The common secret keys are defined as $\vec{s}_* = \sum_{j \in J} \vec{s}_j$ and $\boldsymbol{z}_* = \sum_{j \in J} \boldsymbol{z}_j$.

6.1 Distributed Generation of Evaluation Keys

The distributed generation of evaluation key for threshold version of Algorithm 3 explained in this section is naturally extended to other proposed variants by simple modifications. We omit a description of the LWE switching key which is not the main interest of this paper and is straightforward.

6.1.1 Public Key Generation

The public key for implicit secret keys $\boldsymbol{z}_* = \sum_{j \in J} \boldsymbol{z}_j$ is generated by the following procedures [AJLA+12].

- Each participant $j \in J$ independently generates their own secrets $\vec{s}_j$ and $\boldsymbol{z}_j$.

- Given common random string $\boldsymbol{a}_{\mathtt{crs}}$, each participant calculates $\boldsymbol{b}_j = -\boldsymbol{a}_{\mathtt{crs}} \cdot \boldsymbol{z}_j + \boldsymbol{e}_j$ and shares them to other participants, where $\boldsymbol{e}_j \leftarrow \chi_{\mathtt{err}}$.
- The public key is generated as $\mathtt{pk}^{\mathsf{RLWE}}_{\boldsymbol{z}_*} = (\boldsymbol{a}_{\mathtt{crs}}, \sum_{j\in J} \boldsymbol{b}_j)$.

6.1.2 Generation of Automorphism Keys

The generation of automorphism keys consists of the following two stages.

- Using the shared public key $\mathtt{pk}^{\mathsf{RLWE}}_{\boldsymbol{z}_*}$, each participant generates encryptions $\mathtt{ak}^{Thr}_{j,i} = \mathsf{RLWE}'_{\boldsymbol{z}_*}\left(\boldsymbol{z}_j(X^i)\right)$ as

$$\mathtt{ak}^{Thr}_{j,i} := \left(\mathtt{Enc}^{\mathsf{RLWE}}(B_g^0 \cdot \boldsymbol{z}_j(X^i)), \ldots, \mathtt{Enc}^{\mathsf{RLWE}}(B_g^{d_g-1} \cdot \boldsymbol{z}_j(X^i))\right)$$

  for each i, where $\vec{B_g} = (B_g^0, B_g^1, \ldots, B_g^{d_g-1})$ is a gadget vector. The error for encryption is sampled from $\chi_{\mathtt{smenc}}$, which is a special distribution for a large error to "smudge out" small differences in distributions [AJLA+12], we denote its variance by $\sigma_{\mathtt{smenc}}$ in the later analysis. Next, each participant sends $\mathtt{ak}^{Thr}_{j,i}$ to the computing party.

- The computing party generates automorphism keys $\mathtt{ak}^{Thr}_i$ as follows

$$\mathtt{ak}^{Thr}_i := \sum_{j\in J} \mathtt{ak}^{Thr}_{j,i} = \sum_{j\in J} \mathsf{RLWE}'_{\boldsymbol{z}_*}\left(\boldsymbol{z}_j(X^i)\right) = \mathsf{RLWE}'_{\boldsymbol{z}_*}\left(\boldsymbol{z}_*(X^i)\right).$$

6.1.3 Generation of Blind Rotation Keys

The difference from the generation of the automorphism keys is that, as the sum of components $s_{j,i}$ is done in the exponent, the merging is done by $\mathsf{RGSW} \circledast \mathsf{RGSW}$ multiplications, instead of additions.

- Each participant generates the partial encryption $\mathtt{brk}^{Thr}_{j,i} = \mathsf{RGSW}_{\boldsymbol{z}_*}(X^{s_{j,i}})$ for $i \in [0, n-1]$, where $s_{j,i}$ is the i-th component of $\vec{s_j}$. We can generate the RGSW key using the following equation:

$$\mathtt{brk}^{Thr}_{j,i} := \left(\mathsf{RLWE}'_{\boldsymbol{z}_*}(\boldsymbol{z}_* \cdot X^{s_{j,i}}), \mathsf{RLWE}'_{\boldsymbol{z}_*}(X^{s_{j,i}})\right).$$

  Then, each party sends $\mathtt{brk}^{Thr}_{j,i}$ to the computing party.

- The computing party calculates $\mathtt{brk}^{Thr}_i = \mathsf{RGSW}_{\boldsymbol{z}_*}(X^{s_{*,i}})$ for $i \in [0, n-1]$ using the following equation (note that the error is additive.):

$$\mathtt{brk}^{Thr}_i := \prod_{j\in J} \mathtt{brk}^{Thr}_{j,i} = \prod_{j\in J} \mathsf{RGSW}_{\boldsymbol{z}_*}(X^{s_{j,i}}) = \mathsf{RGSW}_{\boldsymbol{z}_*}(X^{s_{*,i}}).$$

Any party can use these keys to perform secure computations without revealing the secrets of any participants, including the binary gate evaluation [DM15] using the proposed blind rotation technique.

6.2 Performance Analysis

Computational Complexity and Key Size

Following the above evaluation key generation, the computing party finds the evaluation keys:

$$\begin{cases} \mathtt{brk}_i^{Thr} = \mathsf{RGSW}_{\boldsymbol{z}_*}\left(X^{s_{*,i}}\right), & i \in [0, n-1] \\ \mathtt{ak}_u^{Thr} = \mathsf{RLWE}'_{\boldsymbol{z}_*}\left(\boldsymbol{z}_*(X^{g^u})\right), & u \in [1, w] \\ \mathtt{ak}_{-1}^{Thr} = \mathsf{RLWE}'_{\boldsymbol{z}_*}\left(\boldsymbol{z}_*(X^{-g})\right) \end{cases}.$$

Thus, the computation of blind rotation and the structure of the keys are the same as in Algorithm 3. In other words, the computational complexity and key size are the same as in Table 1 in terms of the number of $\odot$ multiplications (computational complexity) and RLWE' ciphertexts (key size).

The number of possible s_i in Gaussian secret is $2 \cdot 12\sigma + 1 \approx 78$, so it is for the case that there are 39 participant with ternary secrets (equivalently 77 participant with binary secrets.) This implies that the proposed blind rotation is preferable for threshold HE as it takes advantage of fast evaluation and the small key size regardless of the secret key distribution.

Error Analysis

The analysis of the blind rotation error for threshold HE assumes that each $\boldsymbol{z}_j$ is a ternary key. This analysis is similar to Section 4, except for the fact that $\mathtt{ak}$ and $\mathtt{brk}$ now have higher error variance. The variance of $\mathtt{pk}_{\boldsymbol{z}_*}^{\mathsf{RLWE}}$ error $\boldsymbol{e}_{\mathtt{pk}}$ is equal to $k\sigma^2$ as it is the sum of errors $\boldsymbol{e}_j$ of all parties. The error of each $\mathsf{RGSW}_{\boldsymbol{z}}(X^{s_{j,i}})$ is equal to $\boldsymbol{v} \cdot \boldsymbol{e}_{\mathtt{pk}} + \boldsymbol{e}_0 \cdot \boldsymbol{z}_* + \boldsymbol{e}_1$. Hence, the error variance is given as $\sigma^2_{\mathtt{fresh}} = \frac{2kN}{3}\sigma^2 + \frac{2kN}{3}\sigma^2_{\mathtt{smenc}} + \sigma^2_{\mathtt{smenc}}$. $\mathsf{RGSW}_{\boldsymbol{z}}\left(X^{s_i}\right)$ is obtained by consecutive multiplication of $\mathsf{RGSW} \circledast \mathsf{RGSW}$ and introduces additive error, whose variance is equal to $\sigma^2_{\mathtt{brk}} = 2k \cdot d_g N \frac{B_g^2}{12} \sigma^2_{\mathtt{fresh}}$. For automorphism keys, again each $\mathsf{RLWE}'_{\boldsymbol{z}}\left(\boldsymbol{z}_j(X^t)\right)$ has the error of variance $\sigma^2_{\mathtt{fresh}}$. Thus the error of $\mathsf{RLWE}'_{\boldsymbol{z}}\left(\boldsymbol{z}(X^t)\right)$ is equal to $\sigma^2_{\mathtt{ak}} = k\sigma^2_{\mathtt{fresh}}$.

The total variance after blind rotation in Algorithm 3 can be estimated as

$$\sigma^2_{\mathtt{acc}} = d_g N \frac{B_g^2}{12} \cdot \left(2n \cdot \sigma^2_{\mathtt{brk}} + \left(\kappa + \frac{N-\kappa}{w}\right) \cdot \sigma^2_{\mathtt{ak}}\right).$$

Similarly, the error variance after Algorithms 4 and 5 can be estimated as

$$\sigma^2_{\mathtt{acc\text{-}me}} = d_g N \frac{B_g^2}{12} \cdot \left((3n+2) \cdot \sigma^2_{\mathtt{brk}} + \left(\kappa + \frac{N-\kappa}{w}\right) \cdot \sigma^2_{\mathtt{ak}}\right),$$
$$\sigma^2_{\mathtt{acc\text{-}ce}} = d_g N \frac{B_g^2}{12} \cdot \left((2n+2) \cdot \sigma^2_{\mathtt{brk}} + \left(\kappa + \frac{N-\kappa}{w}\right) \cdot \sigma^2_{\mathtt{ak}}\right),$$

respectively.

The blind rotation algorithm for AP can be also extended in a similar way, whose blind rotation keys are $\mathsf{RGSW}_{\boldsymbol{z}_*}(Y^{vB_r^t s_{*,i}})$ for $i \in [0, n-1]$, $t \in [0, d_r - 1]$, and $v \in \mathbb{Z}_{B_r}$. Then, the error after AP blind rotation is

$$\sigma^2_{\mathtt{acc}} = d_g N \frac{B_g^2}{12} \cdot \left(2d_r \left(1 - \frac{1}{B_r}\right) n \cdot \sigma^2_{\mathtt{brk}}\right).$$

Since $\sigma_{\mathtt{brk}}$ is much greater than $\sigma_{\mathtt{ak}}$, the proposed blind rotation has less error than the AP variant by exploiting the ring automorphism as well as the less key size. FHEW-like HE parameters are error-sensitive; as our algorithm also produces the least error, our algorithm is more favorable in threshold HE.

7 Conclusion

A new blind rotation technique for homomorphic encryption is proposed with several variants which provide tradeoffs between key size and complexity. We used ring automorphism for scalar multiplication on the exponent, which substitutes the consecutive RGSW multiplications used by the previous AP technique and simplifies the blind rotation procedure. The proposed method offers the best of both previous AP and GINX bootstrapping simultaneously and further improves on them. We demonstrate that our method is better than both approaches in terms of running time and evaluation key size. It offers the additional advantage of reducing the amount of noise introduced during blind rotation, even for the case of the binary key that is the most favorable to GINX.

We also showed a simple threshold HE scheme based on FHEW. This scheme takes advantage of the proposed blind rotation technique since it requires computations under secret keys with distributions wider than binary or ternary. Our analysis showed that the performance and key size could be kept relatively low with increasing the number of participants, unlike GINX blind rotation. This is an important property for the distributed computation settings, which is also supported by AP blind rotation. However, we demonstrated that the proposed blind rotation is better in terms of run time and key size. This demonstrates the high potential of FHEW-like schemes in different applications where the secret key distribution is wider than binary or ternary.

References

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In *International Colloquium on Automata, Languages, and Programming*, pages 403–415. Springer, 2011.

[AJLA$^+$12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT 2012*, pages 483–501, 2012.

[AP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In *CRYPTO 2014*, pages 297–314. Springer, 2014.

[APS15] Martin Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[BD10] Rikke Bendlin and Ivan Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In *TCC 2010*, pages 201–218. Springer, 2010.

[BD20] Zvika Brakerski and Nico Döttling. Hardness of lwe on general entropic distributions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 551–575. Springer, 2020.

[BDF18] Guillaume Bonnoron, Léo Ducas, and Max Fillinger. Large FHE gates from tensored homomorphic accumulator. In *Progress in Cryptology – AFRICACRYPT 2018*, pages 217–251. Springer, 2018.

[BGGJ20] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Chimera: Combining Ring-LWE-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology*, 14(1):316–338, 2020.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[BIP$^+$22] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: Faster FHE instantiated with NTRU and LWE. *Cryptol. ePrint Arch.*, 2022/074, 2022.

[BLP$^+$13] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 575–584, 2013.

[BP16] Zvika Brakerski and Renen Perlman. Lattice-based fully dynamic multi-key FHE with short ciphertexts. In *Advances in Cryptology – CRYPTO 2016*, pages 190–213. Springer, 2016.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, pages 868–886. Springer, 2012.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In *Advances in Cryptology – CRYPTO 2011*, pages 505–524. Springer, 2011.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Multi-key homomorphic encryption from TFHE. In *Advances in Cryptology – ASIACRYPT 2019*, pages 446–472. Springer, 2019.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *Advances in Cryptology – ASIACRYPT 2017*, pages 377–408. Springer, 2017.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CHHS19] Jung Hee Cheon, Minki Hhan, Seungwan Hong, and Yongha Son. A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret LWE. *IEEE Access*, 2019.

[CHI$^+$21] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. In *2021 IEEE Symposium on Security and Privacy (S&P)*, pages 590–607. IEEE, 2021.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 1–19. Springer, 2021.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437. Springer, 2017.

[CM15] Michael Clear and Ciaran McGoldrick. Multi-identity and multi-key leveled FHE from learning with errors. In *Advances in Cryptology – CRYPTO 2015*, pages 630–656. Springer, 2015.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT 2015*, pages 617–640. Springer, 2015.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012/144, 2012.

[GINX16] Nicolas Gama, Malika Izabachene, Phong Q Nguyen, and Xiang Xie. Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems. In *EUROCRYPT 2016*, pages 528–558. Springer, 2016.

[GKPV10] Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Robustness of the learning with errors assumption. 2010.

[HS18] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in HElib. In *Advances in Cryptology – CRYPTO 2018*, pages 93–120. Springer, 2018.

[JP22] Marc Joye and Pascal Paillier. Blind rotation in fully homomorphic encryption with extended keys. In *International Symposium on Cyber Security, Cryptology, and Machine Learning*, pages 1–18. Springer, 2022.

[KDE$^+$21] Andrey Kim, Maxim Deryabin, Jieun Eom, Rakyong Choi, Yongwoo Lee, Whan Ghang, and Donghoon Yoo. General bootstrapping approach for RLWE-based homomorphic encryption. *Cryptol. ePrint Arch.*, 2021/691, 2021.

[KF15] Paul Kirchner and Pierre-Alain Fouque. An improved bkw algorithm for lwe with applications to cryptography and lattices. In *Annual Cryptology Conference*, pages 43–62. Springer, 2015.

[KPZ21] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In *Advances in Cryptology – ASIACRYPT 2021*, pages 608–639. Springer, 2021.

[LHH$^+$21] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE symposium on Security and Privacy (S&P)*, pages 1057–1073. IEEE, 2021.

[LMP21] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. *IACR Cryptol. ePrint Arch.*, 2021/1337, 2021.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):1–35, 2013.

[Mic18] Daniele Micciancio. On the hardness of learning with errors with binary secrets. *Theory of Computing*, 14(1):1–17, 2018.

[MP13] Daniele Micciancio and Chris Peikert. Hardness of SIS and LWE with small parameters. In *Advances in Cryptology – CRYPTO 2013*, pages 21–39. Springer, 2013.

[MP21] Daniele Micciancio and Yuriy Polyakov. Bootstrapping in FHEW-like cryptosystems. In *WAHC'21*, pages 17–28. ACM, 2021.

[MS18] Daniele Miccianco and Jessica Sorrell. Ring packing and amortized FHEW bootstrapping. In *45th International Colloquium on Automata, Languages, and Programming.* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[MW16] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In *Advances in Cryptology – EUROCRYPT 2016*, pages 735–763. Springer, 2016.

[PS16] Chris Peikert and Sina Shiehian. Multi-key FHE from LWE, revisited. In *Theory of Cryptography Conference*, pages 217–238. Springer, 2016.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[TFH] TFHE. Fast fully homomorphic encryption library over the torus. `https://tfhe.github.io/tfhe/`.

[ZZC$^+$21] Tanping Zhou, Zhenfeng Zhang, Long Chen, Xiaoliang Che, Wenchao Liu, and Xiaoyuan Yang. Multi-key fully homomorphic encryption scheme with compact ciphertext. *IACR Cryptol. ePrint Arch.*, 2021/1131, 2021.