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

July 19, 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 are both critical for a number of important applications (like threshold and some multi-key homomorphic encryption (HE) schemes), and provides 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 are less standard and restrict 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. We demonstrate the practicality of the proposed methods by building a prototype implementation within the PALISADE open-source HE library. We illustrate the benefits of our method by providing 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, MP21]: the AP bootstrapping method (originally proposed by Alperin-Sheriff and Peikert [ASP14] 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 [ASP14, 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 (and less standard) 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. In fact, the main attraction of using GINX bootstrapping (with binary or ternary keys, as implemented by [CGGI20, MP21]) remains its lower memory footprint, which is substantially smaller than the AP method.

The use of binary (or even ternary) keys is however quite limiting for two reasons. On the security front, the theoretical 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 (Gaussian with small variance, or ternary) is motivated by practical considerations (limiting error growth during homomorphic computation, and the efficient implementation of GINX bootstrapping) and has some limited theoretical support [MP13, Mic18], but it is still primarily heuristic, a compromise between security and efficiency. In fact, for extreme parameter settings (e.g., sparse keys or a very large number of samples) it is known to weaken the security of the LWE problem [AG11, CHHS19]. For these reasons, current recommendations in [ACC$^+$18] do not support the use of binary keys and suggest the use of ternary keys as a reasonable compromise between practical utility and security.[1]

An even more compelling motivation to use larger secret keys 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 $P_1, \ldots, P_k$, 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 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

[1] To address this, [MP21] offers a simple method to adapt GINX bootstrapping from binary to ternary keys, for a modest (factor 2) penalty in running time, and recent works [KDE$^+$21, BIP$^+$22] have shown that even this extra cost can be eliminated or reduced. (Unfortunately, the methods of [KDE$^+$21, BIP$^+$22] are only useful for ternary keys, and cannot be efficiently extended to larger keys. Thus, this work focus on the more general algorithm in [MP21].)

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.[2])

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$^+$12] (using an arbitrary LWE key generation algorithm for each s_i.) Decryption can also easily be implemented[3] by decoding a ciphertext c using the individual secret key shares, and then adding up the partial decryptions. So, key generation and decryption are minimally interactive.[4] We remark that the threshold schemes [BD10, AJLA$^+$12] predate FHEW-like HE ([BD10] does not explicitly provide any homomorphic computation capability, and [AJLA$^+$12] is a BGV-type encryption scheme.) However, the same principles apply to virtually any LWE-based encryption scheme, including those considered in this paper. Now comes a critical observation: even if the local key shares s_i have binary coefficients, their sum s (used by homomorphic computations and bootstrapping) is no longer binary and has coefficients potentially as large as k, the number of parties participating in the shared decryption protocol. Depending on the application, this number can be quite high, requiring similarly large secret keys. (E.g., see [CHI$^+$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.[5] 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 [ASP14, DM15], with its fast performance (essentially independent of the secret key size) but very large evaluation keys, and
- the GINX bootstrapping method of [GINX16, CGGI20, MP21], with much smaller evaluation keys, but a much 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?

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

[3] This requires some care, adding noise to the partial decryptions to avoid information leakage, as already done in [BD10, AJLA$^+$12].

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

[5] Other methods oriented towards arithmetics on integer or floating-point numbers like [BGV14, CKKS17] offer advantages for a complementary set of applications, but are not within the scope of our paper.

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. The impact of our bootstrapping method become significant with larger keys or 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. (See Figure 2c.) 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": on input the Ring LWE encryption $\mathsf{RLWE}(\boldsymbol{g})$ of a polynomial $\boldsymbol{g}(X)$, a publicly known constant a_i, and an encryption[6] $E(s_i)$ of a secret key coordinate s_i, produce an encryption $\mathsf{RLWE}(\boldsymbol{g} \cdot X^{a_i \cdot s_i})$ of the same polynomial $\boldsymbol{g}$, with its coefficients rotated by $a_i \cdot s_i$ positions. The difference between the two bootstrapping procedures is that

- AP works by including in the evaluation key encryptions $E(a \cdot s_i)$ for all possible values of a and then using a_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 $E(a \cdot s_i)$ for every secret key element s_i.[7]
- 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 $\mathsf{RLWE}(\boldsymbol{g})$ and a modified one $\mathsf{RLWE}(\boldsymbol{g} \cdot X^{a_i})$, using a homomorphic "MUX" gate. This only requires a single encryption $E(s_i)$ for each key element, but it is clearly applicable only to binary secrets. Larger secrets can be handled, for example, writing them in binary, and processing them bit by bit, but at a substantial cost in terms of running time.

In this paper, we present new techniques and optimizations to perform blind rotations using ring automorphisms and key switching, and variants thereof, providing tradeoffs between key size and computation time. The basic idea is to use the automorphisms $\psi_a(X) = X^a$ of the $2N$-th cyclotomic ring $Z[X]/(X^N + 1)$ (for $N = 2^k$, and odd a) and key switching[8] to map encryptions $\mathsf{RLWE}(\boldsymbol{f}(X))$ of $\boldsymbol{f}$ to encryptions $\mathsf{RLWE}(\boldsymbol{f}(X^{a^{-1}}))$ of a permuted polynomial. Then, multiply the result by $E(X^s)$ to get an encryption of $\boldsymbol{f}(X^{a^{-1}}) \cdot X^s$. Finally, use automorphisms/key-switching again to map this ciphertext to an encryption of $\boldsymbol{f}(X) \cdot X^{a \cdot s}$.

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

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

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

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. It is perhaps because of this poor performance that the technique did not receive further attention.

Building on [BDF18], we introduce several variants and optimizations of the technique and show that when properly applied to FHEW it can be quite practical, both in terms of running time and key size. Many of our optimizations revolve around the technical difficulty that automorphisms ψ_a exist only for *odd* coefficients while bootstrapping requires multiplication for both even and odd a. A simple solution to this problem is to express even a's by a sum of two odd numbers (say, 1 and $a-1$.) However, this has the undesirable effect of doubling the number of homomorphic products, the most critical and time-consuming operation required by bootstrapping. As a side note, we remark that, even after pointing out that automorphisms are only applicable for *odd* coefficients a, the bootstrapping algorithm of [BDF18], at least as described in that paper, blindly applies the technique to even and odd coefficients a, which is not correct. In our paper, we formally fix this problem and introduce several optimizations that allow us to deal with arbitrary (even and odd) coefficients a at essentially no added cost.

Remark 1 *In practice, one can use Torus LWE or other similar structures over Ring LWE to achieve additional performance and accelerate computations, as demonstrated in [CGGI20]. However, such schemes apply additional requirements, which may restrict their applicability and choice of parameters. For example, Torus LWE uses floating-point operations, which are efficient with small parameters (such as ring dimension) but require large precision for higher parameters while RLWE may enjoy some acceleration approaches such as CRT decomposition to deal with multi-precision numbers. To preserve generality and 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 homomor-

phic 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.[9]

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 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-

[9] 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.)

Vercauteren (BFV) [Bra12, FV12], were 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 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 $\langle \vec{a}, \vec{b} \rangle$. 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 infinite 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 $m \in \mathbb{Z}$ under the secret key $\vec{s} \leftarrow \chi_{\mathtt{key}}$ 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 $e \leftarrow \chi_{\mathtt{err}}$. 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 $\boldsymbol{z} \leftarrow \chi_{\mathtt{key}}$ 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 $e_i \leftarrow \chi_{\mathtt{err}}$ 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 assume that $(\boldsymbol{t}_0, \cdots, \boldsymbol{t}_{d_r-1})$ is a gadget decomposition of $\boldsymbol{t} \in \mathcal{R}_Q$ if $\boldsymbol{t} = \sum_{i=0}^{d_r-1} g_i \cdot \boldsymbol{t}_i$ where $\vec{g} = (g_0, \ldots, g_{d_r-1})$ is a gadget vector, and $\|\boldsymbol{t}_i\|_\infty < B_r$.

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

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

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

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

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

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

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 $\mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m}_1) \circledast \mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m}_2) \approx \mathsf{RGSW}_{\boldsymbol{z}}(\boldsymbol{m}_1 \cdot \boldsymbol{m}_2)$.

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

Key switching operation converts a ciphertext $\mathsf{RLWE}_{\boldsymbol{z}_1}(\boldsymbol{m})$ encrypted by 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.

Automorphism 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 makes 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.

Secret Key Distribution

The entries of a secret vector for LWE-based scheme and coefficients of a secret polynomial for RLWE-based scheme are chosen from the following secret key distributions as suggested in Homomorphic Encryption Security Standard [ACC+18].
Gaussian (theoretical) Gaussian distribution with standard deviation $\sigma \approx \sqrt{N}$ suggested in [Reg09, LPR13] which guarantees the best theoretical security level close to the uniform secret key distribution
Gaussian (practical) Gaussian distribution with $\sigma \approx 3.19$ which is considered as a more efficient alternative to the Gaussian theoretical distribution
Ternary the uniform distribution with entries from the set $\{-1, 0, 1\}$ which is widely used for lattice-based HE schemes since it introduces a relatively small error
Binary (non-standard) the uniform distribution with entries from the set $\{0, 1\}$ which introduces a small error and leads to the more efficient algorithms but is considered as less secure due to the existing hybrid attacks, e.g., [EJK20]

In general, the secret key distribution is chosen depending on the security level and performance required for a particular scheme [MP21]. Most schemes benefit from using small distributions such

as binary and ternary since they may simplify some algorithms and reduce error accumulation. However, in threshold HE [AJLA+12, MTPBH, Par21, ZZC+21], the secret key distribution should be wider even if the initial secret key distribution is small. Therefore, considering the general case, a large secret key distribution such as Gaussian should also be examined carefully.

2.2 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 RLWE encryption of $\boldsymbol{f} \cdot Y^u$. 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 $\mathtt{brk}_{\boldsymbol{z},\vec{s}}$ corresponding to secrets $\boldsymbol{z}$ and $\vec{s}$, and it outputs the 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 [ASP14] (AP) and [GINX16] (GINX). Both methods rely on the properties of RGSW ciphertext described above.

AP Blind Rotation

In AP blind rotation [DM15, ASP14], 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 initiated 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 $\alpha_i = \sum_{j=0}^{\log_{B_r}(q)-1} a_{i,j} B_r^j$ 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.

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].

Algorithm 1 Blind Rotation: AP [DM15, ASP14]

procedure $\textsc{BlindRotateAP}(\boldsymbol{f}, (\vec{\alpha}, \beta), \{\mathtt{brk}_{i,j,v} = \mathsf{RGSW}_{\boldsymbol{z}}(Y^{vB_r^j s_i})\}_{i,j,v})$

- $\mathtt{acc} \leftarrow (0, \boldsymbol{f} \cdot Y^{\beta})$ ▷ $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{\beta})$
- **for** $(i = 0; i < n; i = i + 1)$ **do** ▷ $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{\beta + \alpha_0 s_0 + \cdots + \alpha_{i-1} s_{i-1}})$
  - **for** $(j = 0; j < \log_{B_r}(q); j = j + 1)$ **do** ▷ $Y^{(\alpha_{i,0} B_r^0 + \cdots + \alpha_{i,j} B_r^j) \cdot s_i}$ is multiplied
    - $\alpha_{i,j} = \left\lfloor \alpha_i / B_r^j \right\rceil \mod B_r$ ▷ $\alpha_i = \sum_{j=0}^{\log_{B_r}(q)-1} \alpha_{i,j} B_r^j$
    - $\mathtt{acc} \leftarrow \mathtt{acc} \circledast \mathtt{brk}_{i,j\alpha_{i,j}}$ ▷ $\mathtt{brk}_{i,j\alpha_{i,j}} = \mathsf{RGSW}_{\boldsymbol{z}}(Y^{\alpha_{i,j} B_r^j s_i})$

return $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{u})$ ▷ $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{\beta + \langle \vec{\alpha}, \vec{s} \rangle})$

In the general case, each secret key element $s_i \in \mathbb{Z}_q$, $i \in [0, N-1]$, is expressed as subset-sum $s_i = \sum_{j=1}^{|U|} u_j \cdot s_{i,j}$ 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 subset U, the blind rotation key is generated as

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

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 of GINX blind rotation is described in Algorithm 2.

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

procedure $\textsc{BlindRotateGINX}(\boldsymbol{f}, (\vec{\alpha}, \beta), \{\mathtt{brk}_{i,j} = \mathsf{RGSW}_{\boldsymbol{z}}(s_{i,j}) \mid s_i = \sum s_{i,j} u_j\})$

- $\mathtt{acc} \leftarrow (0, \boldsymbol{f} \cdot Y^{\beta})$ ▷ $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{\beta})$
- **for** $(i = 0; i < n; i = i + 1)$ **do** ▷ $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{\beta + \alpha_0 s_0 + \cdots + \alpha_{i-1} s_{i-1}})$
  - **for** $(j = 0; j < |U|; j = j + 1)$ **do** ▷ $Y^{\alpha_i \cdot (s_{i,0} u_0 + \cdots + s_{i,j} u_j)}$ is multiplied
    - $\mathtt{acc} \leftarrow \mathtt{acc} + (Y^{\alpha_i u_j} - 1) \cdot (\mathtt{acc} \circledast \mathtt{brk}_{i,j})$ ▷ $Y^{\alpha_i u_j}$ is multiplied if $s_{i,j} = 1$

return $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{u})$ ▷ $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{\beta + \langle \vec{\alpha}, \vec{s} \rangle})$

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 growth significantly with the larger secret key distribution. GINX blind rotation is more efficient than AP for the secret key $\vec{s}$ chosen from small distributions such as binary or ternary secret keys; but less key size in general.

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 key [KDE+21, BIP+22] and cannot be efficiently extended to larger keys.

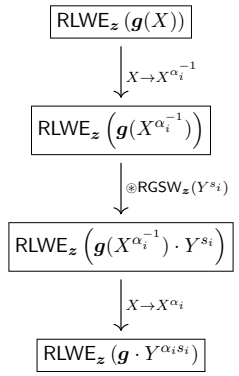


Figure 1: The basic building block of proposed algorithm, where $Y = X^{\frac{2N}{q}}$.

3 New Blind Rotation Techniques

In this section, we present new blind rotation algorithms, which improve on previous methods [ASP14, GINX16, DM15, CGGI17, CGGI20, MP21] in terms of running time, public key size, or both. Recall that blind rotation takes as input an LWE ciphertext $(\vec{\alpha}, \beta)$ and a ring element $\boldsymbol{f} \in \mathcal{R}_Q$, and the goal is to output a ciphertext $\mathsf{RLWE}(\boldsymbol{f} \cdot Y^{\beta + \langle \vec{\alpha}, \vec{s} \rangle}) \in \mathcal{R}_Q^2$ encrypting a rotation of $\boldsymbol{f}$, by a secret amount specified by the decryption of $(\vec{\alpha}, \beta)$ where $Y = X^{\frac{2N}{q}}$. Blind rotation algorithms also make use of a collection of rotation keys $\mathtt{brk}$, which are specific to each blind rotation technique. Our algorithms, just like previous work, use an accumulator $\mathtt{acc} = \mathsf{RLWE}(\boldsymbol{g})$ holding an RLWE encryption of some ring element $\boldsymbol{g}$. This value is initially set to $\boldsymbol{g} = \boldsymbol{f} \cdot Y^{\beta}$, and then multiplied by $\prod_i Y^{\alpha_i s_i}$ for $i = 0, \ldots, n-1$, so that at the end the accumulator contains an encryption $\boldsymbol{f} \cdot Y^{\beta + \langle \vec{\alpha}, \vec{s} \rangle}$ as desired, and we may output $\mathtt{acc}$ as the final result of the blind rotation algorithm. The accumulator is updated through a sequence of $\mathsf{RLWE} \circledast \mathsf{RGSW}$ products, where RLWE is the value of the accumulator, and RGSW is an auxiliary ciphertext holding a secret key element s_i. Our algorithms also make use of ring automorphisms ψ_t and their associated switching keys $\mathtt{ak}_t$, as described in Section 2.

We first describe a base blind rotation algorithm using automorphism with a basic building block and then provide its variants that use auxiliary blind rotation keys to reduce computational complexity. We also present an optimization for $q < 2N$ and $q | 2N$, which uses $Y = X^{\frac{2N}{q}}$ explicitly to improve the complexity of the base algorithm. Finally, we propose several optimizations to reduce the size of evaluation keys.

3.1 The Base Blind Rotation Algorithm

We first give a brief explanation of the basic building block of our algorithms in Figure 1. For a given encryption $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{g}(X))$ corresponding to some intermediate ring element $\boldsymbol{g}(X) \in \mathcal{R}_Q$, we apply automorphism $\mathtt{EvalAuto}_{\alpha_i^{-1}}$ to $\mathtt{acc}$ and obtain $\mathsf{RLWE}_{\boldsymbol{z}}\left(\boldsymbol{g}(X^{\alpha_i^{-1}})\right)$, where α_i^{-1} is computed $\bmod q$. Then, we multiply the accumulator by $\mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_i})$ which results in $\mathsf{RLWE}_{\boldsymbol{z}}\left(\boldsymbol{g}(X^{\alpha_i^{-1}}) \cdot Y^{s_i}\right)$. Finally we apply automorphism $\mathtt{EvalAuto}_{\alpha_i}$ and obtain $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{g} \cdot Y^{\alpha_i s_i})$. This process is repeated for all $i \in [0, n-1]$, so that eventually $\mathtt{acc}$ contains

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

A technical difficulty that needs to be overcome is that automorphisms ψ_{α_i} exists only when α_i is odd. So, we need to give special treatment to even α_i. The full procedure is given in Algorithm 3. Here we briefly discuss how to deal with even coefficients α_i, and a basic optimization that reduces the number of automorphisms by a factor of 2.

Dealing with Even α_i: Since we can apply automorphism $\texttt{EvalAuto}_t$ only for odd t, we 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 blind rotation for the vector $\vec{\omega}$ and obtain $\mathsf{RLWE}\left(\boldsymbol{f}\cdot Y^{\beta+\langle\vec{\omega},\vec{s}\rangle}\right)$. Then we repeatedly multiply $\mathsf{RGSW}\left(Y^{s_i}\right)$ for each even α_i, which results in $\mathsf{RLWE}\left(\boldsymbol{f}\cdot Y^{\beta+\langle\vec{\alpha},\vec{s}\rangle}\right)$[10].

Reducing Number of Automorphisms: Notice that we can combine $\texttt{EvalAuto}_{\alpha_i}$ and $\texttt{EvalAuto}_{\alpha'_{i+1}}$ operations by executing automorphism $\texttt{EvalAuto}_{\alpha_i\alpha'_{i+1}}$, which reduces the number of automorphism operations to n.

Algorithm 3 Blind Rotation: Base Algorithm

procedure $\textsc{BlindRotate}\Big(\boldsymbol{f}, (\vec{\alpha},\beta), \Big(\{\texttt{brk}_i = \mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_i})\}_{i\in[0,n-1]}, \{\texttt{ak}_{2j+1}\}_{j\in[1,q/2-1]}\Big)\Big)$

- **for** $(i=0; i<n; i=i+1)$ **do**
  - **if** α_i is even **then**
    - $\omega_i \leftarrow \alpha_i - 1$
  - **else**
    - $\omega_i \leftarrow \alpha_i$
  - $\omega'_i \leftarrow \omega_i^{-1} \pmod q$
- $\texttt{acc} \leftarrow (\boldsymbol{0}, \left(\boldsymbol{f}\cdot Y^{\beta}\right) \circ (X^{\omega'_0}))$ ▷ $\texttt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f}(X^{\omega'_0})\cdot Y^{\beta\omega'_0})$
- **for** $(i=0; i<n-1; i=i+1)$ **do** ▷ $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{g}(X^{\omega'_i}))$
  - $\texttt{acc} \leftarrow \texttt{acc} \circledast \texttt{brk}_i$ ▷ $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{g}(X^{\omega'_i})\cdot Y^{s_i})$
  - $\texttt{acc} \leftarrow \texttt{EvalAuto}_{\omega_i\omega'_{i+1}}(\texttt{acc}, \texttt{ak}_{\omega_i\omega'_{i+1}})$ ▷ $\mathsf{RLWE}_{\boldsymbol{z}}((\boldsymbol{g}(X)\cdot Y^{\omega_i s_i}) \circ (X^{\omega'_{i+1}}))$
- $\texttt{acc} \leftarrow \texttt{acc} \circledast \texttt{brk}_{n-1}$
- $\texttt{acc} \leftarrow \texttt{EvalAuto}_{\omega_{n-1}}(\texttt{acc}, \texttt{ak}_{\omega_{n-1}})$ ▷ $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f}\cdot Y^{\beta+\langle\vec{\omega},\vec{s}\rangle})$
- **for** $(i=0; i<n; i=i+1)$ **do**
  - **if** α_i is even **then**
    - $\texttt{acc} \leftarrow \texttt{acc} \circledast \texttt{brk}_i$
- ▷ $\texttt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f}\cdot Y^{\beta+\langle\vec{\alpha},\vec{s}\rangle})$

return $\texttt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f}\cdot Y^{u})$

3.2 Reducing Computational Complexity Using Auxiliary Keys

In Algorithm 3, when α_i is even, we multiply the accumulator first by $Y^{\omega_i s_i} = Y^{(\alpha_i-1)s_i}$ and then by Y^{s_i}. Thus, in average, we perform $n/2$ additional $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications, beyond the n products required when all α_i are odd. We show how to get rid of these additional multiplications using a small number of auxiliary blind rotation keys $\mathsf{RGSW}(Y^{s_i+s_{i+1}})$, where by convention $s_n := s_0$.

[10] We can also relax the worst-case condition of Algorithm 3 to the case that at most half of the coefficients of α_i's are even. If the number of even values in $\vec{\alpha}$ is greater than $n/2$, we evaluate $\mathsf{RLWE}(\boldsymbol{f}\cdot Y^{\beta+\langle\vec{\alpha}+\mathbf{1},\vec{s}\rangle})$ using Algorithm 3. Then finally, we obtain $\mathsf{RLWE}(\boldsymbol{f}\cdot Y^{\beta+\langle\vec{\alpha},\vec{s}\rangle})$ by multiplying $\mathsf{RGSW}(Y^{-\Sigma_{i=0}^{n-1}s_i})$ at the end.

The idea is the following. 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 may multiply the accumulator by $Y^{\alpha_i s_i}$ and immediately move to the next step $i+1$. Otherwise, if α_{i+1} is even, we multiply the accumulator by $Y^{\alpha_i(s_i+s_{i+1})}$ instead, and balance this operation by subtracting α_i from α_{i+1}, i.e., we set $\alpha_{i+1} \leftarrow \alpha_{i+1} - \alpha_i$. In either case, the (possibly updated) value of α_{i+1} is odd, preserving the inductive assumption, and we may move to the next iteration. This enables to omit the additional $\mathsf{RLWE} \circledast \mathsf{RGSW}$ multiplications in the last loop of Algorithm 3. The full pseudocode is given in Algorithm 4. Notice that if α_0 is even we can start with any other index i_0 such that α_{i_0} is odd. In the unlikely case that all α_i are even we can start with $\alpha_0 - 1$ and have only one additional multiplication by Y^{s_0} at the very end of the algorithm. However, since the α_i are uniformly random, this happens with exponentially small probability $1/2^n$, and its effect on performance can be safely ignored.

Algorithm 4 Blind Rotation: Computationally Efficient Variant

procedure $\textsc{BlindRotate}\Big(\boldsymbol{f}, (\vec{\alpha}, \beta), \Big(\{\mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_i}), \mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_i+s_{i+1}})\}_{i\in[0,n-1]}, \{\mathtt{ak}_{2j+1}\}_{j\in[1,q/2-1]}\Big)\Big)$

$\vec{\omega} \leftarrow \vec{\alpha}$

$\mathtt{acc} \leftarrow (\mathbf{0}, (\boldsymbol{f} \cdot Y^{\beta}) \circ (X^{\omega'_0}))$ ▷ Assume ω_0 is odd

for $(i = 0; i < n; i = i + 1)$ **do**

  if α_{i+1} is even **then**

    $\mathtt{acc} \leftarrow \mathtt{acc} \circledast \mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_i+s_{i+1}})$ ▷ $Y^{\omega_i(s_i+s_{i+1})}$ is multiplied

    $\omega_{i+1} \leftarrow \omega_{i+1} - \omega_i$ ▷ $Y^{(\omega_{i+1}-\omega_i)s_{i+1}}$ is multiplied later

  else

    $\mathtt{acc} \leftarrow \mathtt{acc} \circledast \mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_i})$

  $\mathtt{acc} \leftarrow \mathtt{EvalAuto}_{\omega_i\omega'_{i+1}}(\mathtt{acc}, \mathtt{ak}_{\omega_i\omega'_{i+1}})$

return $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^{u})$

3.3 Optimization for $q < 2N$

Blind rotations for $q < 2N$ are interesting special cases, as they commonly occur in FHEW-like cryptosystems [DM15, MP21]. When $q < 2N$, our goal is to find $\mathsf{RLWE}(\boldsymbol{f} \cdot Y^{\beta+\langle\vec{\alpha},\vec{s}\rangle}) = \mathsf{RLWE}(\boldsymbol{f} \cdot X^{\frac{2N}{q}\beta+\langle\frac{2N}{q}\vec{\alpha},\vec{s}\rangle})$. Hence, when we have $\mathsf{RGSW}_{\boldsymbol{z}}(X^{s_i})_{i\in[0,n-1]}$ and apply Algorithm 3 as it is, all $\frac{2N}{q}\vec{\alpha}_i$ are even, which is the worst-case in terms of computational complexity, where n additional multiplications of $\mathsf{RGSW}(X^{s_i})$ are required.

With a single additional auxiliary blind rotation key $\mathsf{RGSW}_{\boldsymbol{z}}(X^{-\Sigma_{i=0}^{n-1}s_i})$, we can replace the n multiplications by the $\mathsf{RGSW}(X^{s_i})$'s by a single multiplication. We evaluate $\mathsf{RLWE}(\boldsymbol{f} \cdot X^{\frac{2N}{q}\beta+\langle\vec{\omega},\vec{s}\rangle})$ as in Algorithm 3, where $\omega_i = \frac{2N}{q}\alpha_i + 1$. Since $[\omega_i]_{\frac{2N}{q}} = 1$, we have $[\omega'_i]_{\frac{2N}{q}} = 1$ and $\left[\omega_i\omega'_{i+1}\right]_{\frac{2N}{q}} = 1$ for all i, and $\omega'_i := \omega_i^{-1} \pmod{2N}$. Hence, we only need automorphism keys $\left\{\mathtt{ak}_{\frac{2N}{q}j+1}\right\}_{j\in[1,q-1]}$.

Algorithm 5 Blind Rotation: All-even Case

procedure $\textsc{BlindRotate}\left(\boldsymbol{f}, (\vec{\alpha}, \beta), \left(\{\mathsf{RGSW}_{\boldsymbol{z}}(X^{s_i})\}_{i\in[0,n-1]}, \mathsf{RGSW}_{\boldsymbol{z}}(X^{-\Sigma_{i=0}^{n-1} s_i}), \left\{\mathtt{ak}_{\frac{2N}{q}j+1}\right\}_{j\in[1,q-1]}\right)\right)$

  for $(i = 0; i < n; i = i + 1)$ **do**

    $\omega_i \leftarrow \frac{2N}{q}\alpha_i + 1$

    $\omega_i' \leftarrow \omega_i^{-1} \pmod{2N}$

  $\mathtt{acc} \leftarrow (\boldsymbol{0}, \left(\boldsymbol{f} \cdot X^{\frac{2N}{q}\beta}\right) \circ (X^{\omega_0'}))$

  for $(i = 0; i < n; i = i + 1)$ **do** ▷ $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot X^{\frac{2N}{q}\beta + \langle\vec{\omega},\vec{s}\rangle})$

    $\mathtt{acc} \leftarrow \mathtt{acc} \circledast \mathsf{RGSW}_{\boldsymbol{z}}(X^{s_i})$

    $\mathtt{acc} \leftarrow \mathtt{EvalAuto}_{\omega_i\omega_{i+1}'}(\mathtt{acc}, \mathtt{ak}_{\omega_i\omega_{i+1}'})$

  $\mathtt{acc} \leftarrow \mathtt{acc} \circledast \mathsf{RGSW}_{\boldsymbol{z}}(X^{-\Sigma_{i=0}^{n-1} s_i})$ ▷ $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot X^{\frac{2N}{q}\beta + \frac{2N}{q}\langle\vec{\alpha},\vec{s}\rangle})$

return $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot X^{\frac{2N}{q}u}) = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{f} \cdot Y^u)$

3.4 Key Size Reduction

RLWE′ Blind Rotation Keys Instead of RGSW

In all algorithms so far, we used $\mathsf{RGSW}(Y^{s_i})$ (and $\mathsf{RGSW}(Y^{s_i+s_{i+1}})$) as blind rotation keys. We can replace these keys with smaller RLWE' keys using the compact representation method of [KDE+21], while replacing each $\mathsf{RLWE} \circledast \mathsf{RGSW}$ multiplication with three $\mathcal{R} \odot \mathsf{RLWE}'$ products. Beside the blind rotation keys $\mathsf{RLWE}'_{\boldsymbol{z}}(Y^{s_i})$ (and $\mathsf{RLWE}'_{\boldsymbol{z}}(Y^{s_i+s_{i+1}})$), this method also requires an auxiliary key $\mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{z}^2)$, which, however, remains the same for all s_i (and $s_i + s_{i+1}$). So, its impact on blind key storage requirements is minor. As each RGSW ciphertext is composed of two RLWE' ciphertexts and each $\mathsf{RLWE} \circledast \mathsf{RGSW}$ multiplication requires two $\mathcal{R} \odot \mathsf{RLWE}'$ products, this method offers a memory-computation trade-off: key storage is roughly cut in half, and computation time is increased by a factor 3/2. In typical HE application scenarios, key size reduction is crucial as it lowers the computation and communication cost for resource-limited clients. So, the increase in running time may be affordable.

The compact multiplication method works as follows. Given the current accumulator $\mathtt{acc} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{g}) = (\boldsymbol{a}, \boldsymbol{b})$, auxiliary key $\mathsf{RLWE}'_{\boldsymbol{z}}(Y^{s_i})$, and compact blind rotation key $\mathsf{RLWE}'_{\boldsymbol{z}}(Y^{s_i})$, we first multiply $\mathsf{RLWE}'_{\boldsymbol{z}}(Y^{s_i})$ by the ring elements $\boldsymbol{a}$ and $\boldsymbol{b}$ to obtain

$$\begin{aligned}\boldsymbol{a} \odot \mathsf{RLWE}'_{\boldsymbol{z}}(Y^{s_i}) &= \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{a} \cdot Y^{s_i}) = (\boldsymbol{a}', \boldsymbol{b}')\\ \boldsymbol{b} \odot \mathsf{RLWE}'_{\boldsymbol{z}}(Y^{s_i}) &= \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{b} \cdot Y^{s_i})\end{aligned}$$

Then we compute $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{a} \cdot \boldsymbol{z} \cdot Y^{s_i})$ by evaluating

$$\boldsymbol{a}' \odot \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{z}^2) + (\boldsymbol{b}', 0) = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{a}' \cdot \boldsymbol{z}^2 + \boldsymbol{b}' \cdot \boldsymbol{z}) = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{a} \cdot \boldsymbol{z} \cdot Y^{s_i}).$$

Finally, we add $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{a} \cdot \boldsymbol{z} \cdot Y^{s_i})$ and $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{b} \cdot Y^{s_i})$ together to obtain the updated $\mathtt{acc}$:

$$\begin{aligned}\mathtt{acc} &\leftarrow \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{a} \cdot \boldsymbol{z} \cdot Y^{s_i}) + \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{b} \cdot Y^{s_i})\\ &= \mathsf{RLWE}_{\boldsymbol{z}}((\boldsymbol{a} \cdot \boldsymbol{z} + \boldsymbol{b}) \cdot Y^{s_i}) = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{g} \cdot Y^{s_i}).\end{aligned}$$

In summary, we can replace all $\mathsf{RLWE} \circledast \mathsf{RGSW}$ products in Algorithms 3, 4, and 5 with the alternative $\textsc{AltMult}$ multiplication procedure described in Algorithm 6.

Algorithm 6 Alternative Multiplication with RLWE' Keys [KDE+21]

procedure $\textsc{AltMult}(\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{m}_1), \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{m}_2), \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{z}^2))$

$(\boldsymbol{a}, \boldsymbol{b}) \leftarrow \mathsf{RLWE}(\boldsymbol{m}_1)$

$(\boldsymbol{a}', \boldsymbol{b}') \leftarrow \boldsymbol{a} \odot \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{m}_2)$

$\mathtt{res} \leftarrow \boldsymbol{a}' \odot \mathsf{RLWE}'_{\boldsymbol{z}}(\boldsymbol{z}^2) + (\boldsymbol{b}', 0)$ ▷ $\mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{a}' \cdot \boldsymbol{z}^2 + \boldsymbol{b}' \cdot \boldsymbol{z}) = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{a} \cdot \boldsymbol{z} \cdot \boldsymbol{m}_2)$

$\mathtt{res} \leftarrow \mathtt{res} + \boldsymbol{b} \odot \mathsf{RLWE}'(\boldsymbol{m}_2)$ ▷ $\mathtt{res} = \mathsf{RLWE}_{\boldsymbol{z}}((\boldsymbol{a} \cdot \boldsymbol{z} + \boldsymbol{b}) \cdot \boldsymbol{m}_2)$

return $\mathtt{res} = \mathsf{RLWE}_{\boldsymbol{z}}(\boldsymbol{m}_1 \cdot \boldsymbol{m}_2)$

Key size and Computation Tradeoffs with Fewer Automorphism Keys

We do not have to carry all the switching keys. $\mathtt{EvalAuto}_k(\cdot, \mathtt{ak}_k)$ for an odd k can be replaced with sequence of operations $\mathtt{EvalAuto}_{k_j}(\cdot, \mathtt{ak}_{k_j})$, where $k = \prod_j k_j$. In other words,

$$\mathtt{EvalAuto}_k(\mathtt{ct}, \mathtt{ak}_k) = \mathtt{EvalAuto}_{k_0}(\mathtt{EvalAuto}_{k_1}(\cdots \mathtt{EvalAuto}_{k_{i-1}}(\mathtt{ct}, \mathtt{ak}_{k_{i-1}}) \cdots), \mathtt{ak}_{k_1}), \mathtt{ak}_{k_0}).$$

Thus, we only need to carry a set of keys $\{k_j\}_{j \in [0, i-1]}$ whose subset-products generates $\mathbb{Z}_q^*$.

As an extreme example, all the rotations is done using only two switching keys $\mathtt{ak}_5$ and $\mathtt{ak}_{-1}$ by applying (or not) $\mathtt{EvalAuto}_{-1}(\mathtt{ct}, \mathtt{ak}_{-1})$ and (at most $q-1$) $\mathtt{EvalAuto}_5(\mathtt{ct}, \mathtt{ak}_5)$. Here, we use that $\{-1, 5\}$ generates $\mathbb{Z}_q^*$, where q is a power of two. For another example, using $\log q$ keys $\mathtt{ak}_{-1}, \mathtt{ak}_5, \mathtt{ak}_{5^2}, \mathtt{ak}_{5^4}, \ldots, \mathtt{ak}_{5^{q/2}}$, we can perform the same operation as $\mathtt{EvalAuto}_k$ (for any odd k) with at most $\log q$ of $\mathtt{EvalAuto}$ with keys from the given set.

4 Analysis

4.1 Complexity and Key Size Analysis

The comparisons of computational complexity and key size are given in Table 1. Both settings are directly motivated by applications. In particular, $q = 2N$ is used in general bootstrapping of RLWE-based HE [KDE+21], while $q < 2N$ often arises in functional bootstrapping of FHEW-like HE [DM15, CGGI17, CGGI20, MP21].

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. We note that each $\mathsf{RLWE} \circledast \mathsf{RGSW}$ product requires two $\mathcal{R} \odot \mathsf{RLWE}'$ products, while key switching is performed with a single $\mathcal{R} \odot \mathsf{RLWE}'$ multiplication. So, both $\mathsf{RLWE} \circledast \mathsf{RGSW}$ products and key switching operations are easily expressed in terms of $\mathcal{R} \odot \mathsf{RLWE}'$ products. Another common measure of complexity used in previous work on FHEW-like HE is the number of NTT performed by the algorithms. We note that one can easily convert the number of $\mathcal{R} \odot \mathsf{RLWE}'$ products to the number of NTT as each $\mathcal{R} \odot \mathsf{RLWE}'$ requires precisely $(d_g + 1)$ NTT operations. As a side remark, the cost of BGV/BFV/CKKS bootstrapping is often measured in terms of rotation and relinearization operations (a form of key switching), which is also easily expressed in terms of $\mathcal{R} \odot \mathsf{RLWE}'$ products.

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 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.

Previous Methods

AP The complexity and key size of the AP technique depends on the value of the gadget decomposition base B_r used to decompose α_i into a sequence of small integers $\alpha_{i,j}$, where $\alpha_i = \sum_{j=0}^{\log_{B_r} q-1} \alpha_{i,j}B_r^j$ and $d_r = \lceil \log_{B_r} q \rceil$. The blind rotation key for AP is given by $\left\{\mathsf{RGSW}_{\boldsymbol{z}}(Y^{vB_r^j s_i})\right\}_{i,j,v}$, where $(i,j,v) \in [0,n-1]\times[0,d_r-1]\times[1,B_r-1]$. (Note that ciphertexts for $v=0$ are trivial and are not needed.) So, the key is composed by $d_r(B_r-1)n$ RGSW ciphertexts, or, equivalently, $2d_r(B_r-1)n$ RLWE' ciphertexts. In the worst-case, $d_r\mathsf{RLWE}\circledast\mathsf{RGSW}$ is required to multiply $Y^{\alpha_{i,j}B_r^j}$ for each $i=0,\ldots,n-1$ and $j=0,\ldots,d_r-1$ to $\mathtt{acc}$. So, the worst-case complexity is $2d_rn$ of $\mathcal{R}\odot\mathsf{RLWE}'$ multiplications. However there is a $1/B_r$ chance for each $\alpha_{i,j}$ to be 0, in which case the multiplication by $\mathsf{RGSW}_{\boldsymbol{z}}(Y^{\alpha_{i,j}B_r^j s_i})$ can be skipped, reducing the the average-case complexity to $2d_r\left(1-\frac{1}{B_r}\right)n$ $\mathcal{R}\odot\mathsf{RLWE}'$ multiplications.

GINX The complexity and key size of the GINX technique [MP21] depends on the cardinality of the set U used to represent each $s_i = \sum_{u_j\in U} s_{i,j}u_j$. We note that the size of this set can be set optimally to $|U| = \lceil \log(\|\boldsymbol{s}\|_\infty)\rceil$. The blind rotation key $\{\mathsf{RGSW}_{\boldsymbol{z}}(s_{i,j})|s_i = \sum s_{i,j}u_j\}$ is composed of $|U|n$ RGSW ciphertexts, or, equivalently, $2|U|n$ RLWE' ciphertexts. The algorithm also requires $2|U|n$ $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications by $Y^{\alpha_i s_{i,j}u_j}$ for each $i=0,\ldots,n-1$ and $u_j\in U$ to $\mathtt{acc}$. In GINX optimization for ternary keys [KDE+21, BIP+22], the number of $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications is reduced from $4n$ to $2n$. To distinguish from original GINX method we denoted this optimization *GINX**.

Proposed Methods

The Base Blind Rotation (Algoritm 3) In our Algorithm 3, the public key for blind rotation contains n RGSW ciphertexts ($\{\mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_i})\}_{i\in[0,n-1]}$) and $q/2-1$ RLWE' ciphertexts ($\{\mathtt{ak}_{2j+1}\}_{j\in[1,q/2-1]}$). Algorithm 3 performs n $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications and n $\mathcal{R}\odot\mathsf{RLWE}'$ multiplications to multiply the accumulator by $Y^{\omega_i\cdot s_i}$; for an even α_i, we have to multiply $\mathsf{RGSW}(Y^{s_i})$, and thus we require n and $3n/2$ $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications in the worst-case and average-case, respectively. Algorithm 3 is denoted by "Ours(Alg. 3)" in Table 1.

Reducing Computational Complexity Using Auxiliary Keys (Algoritm 4) In Algorithm 4, we need auxiliary blind rotation keys $\mathsf{RGSW}(Y^{s_i+s_{i+1}})$, so we need a total of $2n$ of RGSW ciphertexts and $q/2-1$ of RLWE' ciphertexts. This allows omitting the additional $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications by $\mathsf{RGSW}(Y^{s_i})$ at the end of the algorithm. So, the numbers of operations is n $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications and n $\mathcal{R}\odot\mathsf{RLWE}'$ multiplications on average and $n+1$ $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications and n $\mathcal{R}\odot\mathsf{RLWE}'$ multiplications in the all-even case, which occurs with probability $\frac{1}{2^n}$. Hence, Algorithm 4 is faster than the base case in Algorithm 3, and has constant execution time, at the cost of larger blind rotation keys. The proposed reduced-complexity variant is shown as "Ours(Alg. 4)" in Table 1.

Table 1: Complexity and key size of each blind rotation technique. Key size (# keys) is the number of RLWE' ciphertexts, and computational complexity (# mult) is the number of $\mathcal{R} \odot \mathsf{RLWE}'$.

Method	Using RGSW keys			Using RLWE' keys		
	# keys	# mult		# keys	# mult	
		average	worst		average	worst
AP [ASP14, DM15]	$2d_r(B_r-1)n$	$2d_r\left(1-\frac{1}{B_r}\right)n$	$2d_rn$	$d_r(B_r-1)n+1$	$3d_r\left(1-\frac{1}{B_r}\right)n$	$3d_rn$
GINX [GINX16, CGGI20, MP21]	$2\|U\|n$	$2\|U\|n$	$2\|U\|n$	$\|U\|n+1$	$3\|U\|n$	$3\|U\|n$
Ours (Alg. 3)	$2n+q/2-1$	$4n$	$5n$	$n+q/2$	$5.5n$	$7n$
Ours (Alg. 4)	$4n+q/2-1$	$3n+\frac{2}{2^n}$	$3n+2$	$2n+q/2$	$4n+\frac{3}{2^n}$	$4n+3$
Ours (Alg. 5)	$2n+q+1$	$3n+2$	$3n+2$	$n+q+1$	$4n+3$	$4n+3$

Optimization for $q < 2N$ (Algoritm 5) When $q < 2N$, the variant in Algorithm 5 is efficient in terms of key size and computational complexity. It requires $(n+1)$ RGSW ciphertexts for $\{\mathsf{RGSW}_{\boldsymbol{z}}(X^{s_i})\}_{i\in[0,n-1]}$ and $\mathsf{RGSW}_{\boldsymbol{z}}(X^{\Sigma_{i=0}^{n-1}s_i})$, and $(q-1)$ RLWE' ciphertexts for the switching keys $\left\{\mathtt{ak}_{\frac{2N}{q}j+1}\right\}_{j\in[1,q-1]}$, for a total of $2n+q+1$ RLWE' ciphertexts. The computational cost is $(n+1)$ $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications, and n $\mathcal{R}\odot\mathsf{RLWE}'$ multiplications for key switching. Our Algorithm 5 for the all-even case is denoted by "Ours(Alg. 5)" in Table 1.

Key Size Reduction Using RLWE' Keys (Algorithm 6) As shown in subsection 3.4, we can replace RGSW ciphertexts in blind rotation keys by RLWE' ciphertexts. Then, the key size for RGSW blind rotation keys is reduced to almost half. As a tradeoff, $\mathsf{RLWE}\circledast\mathsf{RGSW}$ is replaced by $\textsc{AltMult}$ in Algorithm 6, which involves three $\mathcal{R}\odot\mathsf{RLWE}'$ multiplications instead of two. This modification can be applied to our proposed methods as well as AP and GINX. The comparison with key size reduction using RLWE' is given in separate columns in Table 1.

4.2 Error Analysis

We present a theoretical analysis of the error growth to set the proper auxiliary modulus and show the implementation results with concrete parameters. We use an approach from [DM15, MP21] to estimate the variance $\sigma^2_{\mathtt{acc}}$ from the blind rotation procedure. The total error estimation for algorithms using blind rotation such as FHEW/TFHE bootstrapping [DM15, MP21, CGGI20], amortized FHEW bootstrapping [MS18], and general bootstrapping for RLWE HE [KDE+21], can be easily calculated using this value.

The error variance introduced by single $\mathcal{R}\odot\mathsf{RLWE}'$ 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 $\mathcal{R}\odot\mathsf{RLWE}'$ multiplication. For sake of brevity we denote $\sigma^2_{\odot} := d_gN\frac{B_g^2}{12}\sigma^2$. The variances $\sigma^2_{\mathtt{acc}-AP}$ and $\sigma^2_{\mathtt{acc}-GINX}$ can be estimated as [MP21]

$$\sigma^2_{\mathtt{acc}-AP} = 2d_rn\cdot\sigma^2_{\odot} \text{ and } \sigma^2_{\mathtt{acc}-GINX} = 4|U|n\cdot\sigma^2_{\odot}.$$

In Algorithms 3, 4, and 5, each $\mathtt{acc}\circledast\mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_i})$ introduces additive error whose variance is $2\cdot\sigma^2_{\odot}$, as Y^{s_i} is a monomial. The automorphism operation $\psi_{\omega_i\omega'_{i+1}}$ is error free, and key switching introduces additive error with variance $\sigma^2_{\odot}$. Hence, $\mathtt{EvalAuto}_{\omega_i\omega'_{i+1}}(\mathtt{acc},\mathtt{ak}_{\omega_i\omega'_{i+1}})$ introduces additive error with variance $\sigma^2_{\odot}$. In the worst-case, Algorithm 3 requires $2n$ $\mathsf{RLWE}\circledast\mathsf{RGSW}$ multiplications and n $\mathcal{R}\odot\mathsf{RLWE}'$ multiplications; thus the error after blind rotation is given as $5n\cdot\sigma^2_{\odot}$. Similarly, the error after Algorithms 4 and 5 has variance $3(n+2)\cdot\sigma^2_{\odot}$.

Table 2: Estimation of error growth for each blind rotation technique. The value is normalized to the common multiplier $\sigma_{\odot}^2$, the variance of error introduced by an $\mathcal{R} \odot \mathsf{RLWE}'$.

Method	$\sigma_{\mathsf{acc}}^2/\sigma_{\odot}^2$	
	Using RGSW keys	Using RLWE′ keys
AP [ASP14, DM15]	$2d_rn$	$(\|\boldsymbol{z}\|^2+2)d_rn$
GINX [GINX16, CGGI20, MP21]	$4\|U\|n$	$2(\|\boldsymbol{z}\|^2+2)\|U\|n$
Ours (Alg. 3)	$5n$	$(2\|\boldsymbol{z}\|^2+5)n$
Ours (Alg. 4)	$3n+2$	$(\|\boldsymbol{z}\|^2+3)n+(\|\boldsymbol{z}\|^2+3)$
Ours (Alg. 5)	$3n+2$	$(\|\boldsymbol{z}\|^2+3)n+(\|\boldsymbol{z}\|^2+3)$

AltMult in Algorithm 6 introduces a larger error than $\mathsf{RLWE} \circledast \mathsf{RGSW}$; it presents error with variance $(\|\boldsymbol{z}\|^2+2)\cdot\sigma_{\odot}^2$. We note that $\|\boldsymbol{z}\|$ can be assumed to be bounded by $\sqrt{N/2}$ when $\boldsymbol{z}$ follows ternary uniform distribution. Table 2 summarizes the estimation of error growth for all algorithms. We remark that $|U| = \lceil \log \|s\|_{\infty} \rceil$, e.g., $|U| = 2$ for ternary secret.

4.3 Comparison by Key Distribution

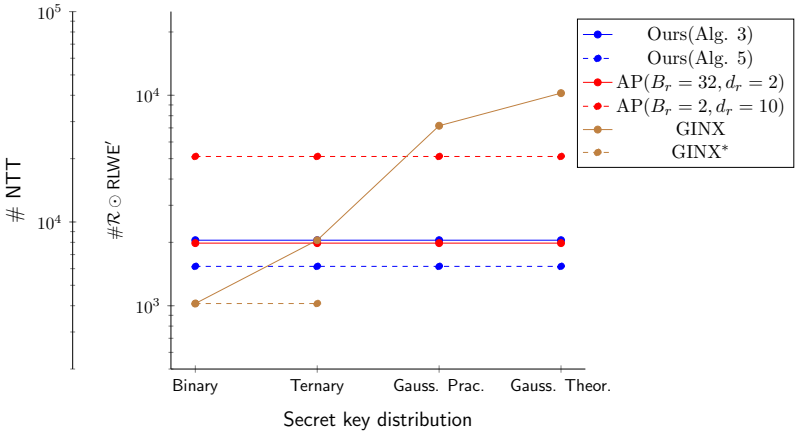
The computational complexity and key sizes for each key distribution are given in Figures 2a and 2b. The figures are for the parameter set STD128, $(n, q, N, Q) = (512, 1024, 1024, 2^{27})$, proposed by Micciancio and Polyakov [MP21] based on HE Security Standard [ACC+18], which is for the 128-bit security w.r.t. classical computer attacks for ternary key distribution. Here, "Gauss. Pract." corresponds to the Gaussian secret key distribution with standard deviation $\sigma = 3.19$; "Gauss. Theor." to the case $\sigma = \sqrt{n}$; both secret key distributions truncated using a standard 12σ tail bound.

It can be seen that the proposed algorithm has less computational complexity and key size compared to AP and GINX when the key distribution is wider than binary (except for GINX*) and the least error for all key distributions. We also demonstrate that our method is getting more efficient than existing blind rotation techniques in situations when binary and ternary secret key distributions cannot be used. For example, the key distribution wider than ternary is used to construct the threshold variant of the threshold HE schemes introduced in [AJLA+12, ZZC+21], where secret keys are accumulated from partial secret keys.

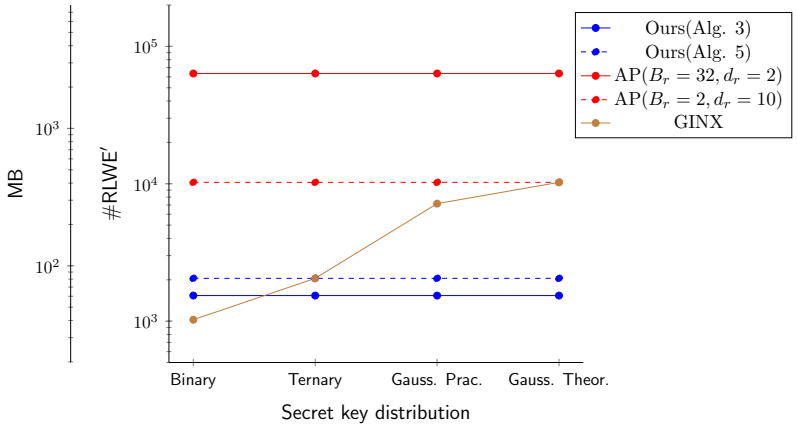
5 Implementation

In this section, we present the implementation results of the proposed blind rotation on FHEW bootstrapping as a useful application. Also, we compare it to the AP and GINX blind rotation techniques. We use the gate bootstrapping of FHEW [DM15] implemented in PALISADE [PAL21]. According to the theoretical analysis presented early, similar results will be achieved for TFHE [CGGI20] by using floating-point operations and DFTs instead of operation over finite rings and NTTs respectively for each discussed blind rotation technique.

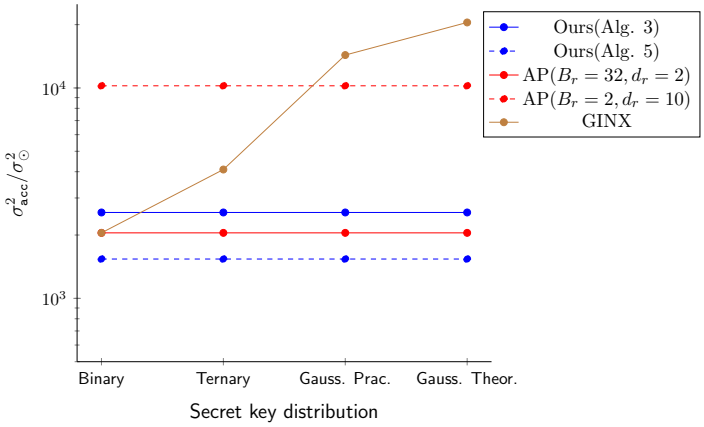
We follow the approach from [MP21] for our experiment, where several parameter sets are introduced based on different security levels. We omit the details on parameter selection as it can be found directly in [MP21]. We compared accumulated errors for different techniques discussed in this paper.



(a) Comparison of blind rotation computational complexity in terms of number of $\mathcal{R} \odot \mathsf{RLWE}'$ operations. One $\mathcal{R} \odot \mathsf{RLWE}'$ operation is roughly $(d_g + 1)$ of NTT operations.



(b) Comparison of blind rotation key size in terms of number of RLWE' elements. One RLWE' element is roughly $2d_g N \log Q$-bit of space.



(c) Comparison of blind rotation error variance. The value is normalized to the common multiplier $\sigma^2_{\odot} = d_g N \frac{B_g^2}{12} \sigma^2$.

Figure 2: Comparison of blind rotation computational complexity, key size, and error.

Table 3: Parameter sets and experimental error variances of blind rotation technique for ternary secret distribution. For AP, we use $d_r = 2$. For Ours, we use Alg. 4

Parameter set	n	q	N	$\log_2 Q$	B_g	σ^2_{acc}		
						AP	GINX	Ours
STD128	512	1024	1024	27	2^7	1.00×10^{11}	1.78×10^{11}	$\mathbf{5.92 \times 10^{10}}$
STD192	1024	1024	2048	37	2^{13}	9.14×10^{14}	2.02×10^{15}	$\mathbf{8.09 \times 10^{14}}$
STD256	1024	2048	2048	29	2^8	1.41×10^{12}	3.13×10^{12}	$\mathbf{9.95 \times 10^{11}}$
STD128Q	1024	1024	2048	50	2^{25}	1.53×10^{22}	3.35×10^{22}	$\mathbf{1.18 \times 10^{22}}$
STD192Q	1024	1024	2048	35	2^{12}	2.68×10^{14}	5.26×10^{14}	$\mathbf{1.88 \times 10^{14}}$
STD256Q	2048	1024	2048	27	2^7	7.07×10^{11}	1.46×10^{12}	$\mathbf{5.77 \times 10^{11}}$

Table 4: Timing results (in [ms]) of blind rotation keys for FHEW bootstrapping (NAND gate).

Key distribution	Ternary					Gaussian (practical, $\sigma = 3.19$)				
Parameter set	AP	GINX	GINX*	Ours		AP	GINX	GINX*	Ours	
				Alg. 4	Alg. 5				Alg. 4	Alg. 5
STD128	127	129	86	**104**	**103**	127	801	-	**104**	**103**
STD192	452	466	315	**386**	**379**	452	2831	-	**386**	**379**
STD256	513	544	359	**427**	**433**	513	3283	-	**427**	**433**
STD128Q	341	353	237	**308**	**310**	341	2096	-	**308**	**310**
STD192Q	433	448	294	**377**	**373**	433	2772	-	**377**	**373**
STD256Q	1061	1074	759	**883**	**870**	1061	6612	-	**883**	**870**

5.1 Parameter Sets and Blind Rotation Error

Table 3 shows the error variances of ciphertexts after applying different blind rotation techniques by the parameter sets for FHEW presented in [MP21] and [Ope22]. For the names of parameter sets, we follow the notation from [MP21]. The prefix "STD" denotes the parameter set that corresponds to the HE security standard [ACC+18]. The numbers in the parameter set names correspond to the estimated bits of security, and the suffix "Q" denotes the parameter sets secure against quantum attack. For all parameter sets the ternary secret key distribution is used.

The variances of blind rotation error in Table 3, σ^2_{acc} was obtained numerically. For each parameter set, we record the error values after blind rotation for 500 trials and then estimate the value of σ^2_{acc}. The proposed blind rotation technique is implemented using PALISADE. We use AP and GINX implementation provided in PALISADE library [PAL21] as it is.

Table 3 demonstrates that the error variance after blind rotation by Algorithm 5 is reduced by $11\% - 30\%$ and $60\% - 68\%$ compared to AP and GINX, respectively. This experimental result meets the theoretical analysis in Table 2.

The total error and decoding failure probability can be calculated from the given σ_{acc}. We refer to [MP21] for the detailed calculation as the other parts of errors are common regardless of the blind rotation technique used. However, the error after blind rotation takes a major portion of the error in the whole bootstrapping process. The achieved error variance shows an improvement in comparison with previous methods, and thus it reduces the decryption failure probability.

Table 5: Blind rotation key size (in [MB]) for FHEW bootstrapping (NAND gate).

Key distribution	Ternary					Gaussian (practical, $\sigma = 3.19$)				
Parameter set	AP	GINX	GINX*	Ours		AP	GINX	GINX*	Ours	
				Alg. 4	Alg. 5				Alg. 4	Alg. 5
STD128	1674	54	54	**67.5**	**54.0**	1674	189	-	**67.5**	**54.0**
STD192	6682	222	222	**249.7**	**166.6**	6682	777	-	**249.7**	**166.6**
STD256	10440	232	232	**289.9**	**232.1**	10440	812	-	**289.9**	**232.1**
STD128Q	6200	200	200	**225.0**	**150.0**	6200	700	-	**225.0**	**150.0**
STD192Q	6510	210	210	**236.2**	**157.6**	6510	756	-	**236.2**	**157.6**
STD256Q	9720	216	216	**269.9**	**216.1**	9720	756	-	**269.9**	**216.1**

5.2 Runtime Results

The evaluation environment was with PALISADE v.1.11.7 in Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz, running Ubuntu 16.04.7 LTS. We compiled with gcc 8.4.0 and the following CMake flags: NATIVE_SIZE=64. We note that the current version of PALISADE does not contain AVX-optimized implementation which can speed up all bootstrapping algorithms (see [MP21] for more details). To provide an impartial comparison of bootstrapping algorithms, we have implemented all of them using the identical library and computing environment.

Tables 4 and 5 contain experimental results for NAND gate evaluation in FHEW in terms of runtime and blind rotation key size, respectively. We provide experiments for different parameter sets for ternary and practical Gaussian secret key distributions. In our experiments we used Algorithms 4 and 5 which are the most suitable for the parameters in Table 3 among the proposed techniques. We follow [MP21] for GINX with ternary and Gaussian secret keys and GINX* with ternary secret keys. We do not provide GINX* experimental results for Gaussian secret keys, as GINX* is an optimization only for ternary secret keys. In Table 5 we consider only the key sizes for the blind rotation keys. LWE switching keys are also required for the FHEW bootstrapping. However, these keys are the same for each blind rotation technique we analyzed, and thus we omitted them in Table 5.

Considering that the gate bootstrapping includes additional operations other than blind rotation, such as LWE addition and key switching, the results in Tables 4 and 5 roughly meet the theoretical analysis conducted in Section 4. Table 4 shows that both Algorithms 4 and 5 improve the runtime for the gate bootstrapping by about 15%, compared to AP for both ternary and Gaussian secret key. For ternary secret key, our method is also about 17% faster than GINX but about 23% slower than GINX*. However, for Gaussian secret key our methods surpass GINX by more than 7.4 times and GINX* is not efficiently scalable for large secret key distributions. Table 5 shows that Algorithm 5 gives up to 45 times reduced key size compared to AP, and about 25% reduction compared to GINX in case of ternary. The key size of Algorithm 4 is up to 36 times smaller than that of AP.

6 Applications to Threshold Homomorphic Encryption

The above analysis demonstrates that 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 base 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 Algorithms 4 and 5 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 [AJLA+12]

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

- Each participant $j \in J$ independently generates their own secret $\vec{s}_j$ 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 all $i \in \{2\ell + 1 : 1 \le \ell \le q/2 - 1\}$, 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 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 $\mathtt{brk}_{j,i}^{Thr} = \mathsf{RGSW}_{\boldsymbol{z}_*}(Y^{s_{j,i}})$ 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 following equation:

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

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

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

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

Any party can use these keys to perform secure computations without revealing 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:

$$\left\{\mathtt{brk}_i^{Thr} = \mathsf{RGSW}_{\boldsymbol{z}_*}(Y^{s_{*,i}})\right\}_{i \in [0,n-1]}, \left\{\mathtt{ak}_{2i+1}^{Thr} = \mathsf{RLWE}'_{\boldsymbol{z}_*}(\boldsymbol{z}_*(X^{2i+1}))\right\}_{i \in [1,q/2-1]}.$$

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 $\mathcal{R} \odot \mathsf{RLWE}'$ multiplications (computational complexity) and RLWE' ciphertexts (key size).

The X-axis of Figure 2 (key distribution) can be heuristically replaced by the number of participants of threshold HE. For example, the number of possible s_i in "Gauss. Prac." is $2 \cdot 12\sigma + 1 = 77.56$, 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. Practically, the optimized parameter differs by the number of participants as the blind rotation error (see analysis below) depends on the number of participants, but the trend will follow Figure 2.

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 4.2, 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 sum of errors $\boldsymbol{e}_j$ of all parties. The

error of each $\mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_{j,i}})$ is equal to $\boldsymbol{v}\cdot\boldsymbol{e}_{\mathtt{pk}}+\boldsymbol{e}_0\cdot\boldsymbol{z}_*+\boldsymbol{e}_1$ so the variance can be calculated as $\sigma^2_{\mathtt{fresh}}=\frac{2kN}{3}\sigma^2+\frac{2kN}{3}\sigma^2_{\mathtt{smenc}}+\sigma^2_{\mathtt{smenc}}$. $\mathsf{RGSW}_{\boldsymbol{z}}(Y^{s_i})$ is obtained by consecutive multiplication of $\mathsf{RGSW}\circledast\mathsf{RGSW}$ and introduces additive error, whose variance is equal to $\sigma^2_{\mathtt{brk}}=2k\cdot d_gN\frac{B_g^2}{12}\sigma^2_{\mathtt{fresh}}$. For automorphism keys, we again have that 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 is equal to

$$\sigma^2_{\mathtt{acc}}=d_gN\frac{B_g^2}{12}\cdot\left(3n\cdot\sigma^2_{\mathtt{brk}}+n\cdot\sigma^2_{\mathtt{ak}}\right).$$

Similarly, the error variance after Algorithms 4 and 5 are the same as

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

The blind rotation algorithm for AP can be also extended in the similar way, whose blind rotation keys are $\mathsf{RGSW}_{\boldsymbol{z}_*}(Y^{vB_r^ts_{*,i}})$ 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

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

Since $\sigma_{\mathtt{brk}}$ is much greater than $\sigma_{\mathtt{ak}}$, the proposed Algorithm 3 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 previous techniques 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 except for the isolated case of binary keys. 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 advantages 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 complexity, key size, and error. This demonstrates the high potential of FHEW-like schemes in different applications where the secret key distribution is wider than binary or ternary.

References

[ACC+18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[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.

[ASP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In *CRYPTO 2014*, pages 297–314. Springer, 2014.

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

[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.

[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.

[EJK20] Thomas Espitau, Antoine Joux, and Natalia Kharchenko. On a dual/hybrid approach to small secret LWE. In *Progress in Cryptology – INDOCRYPT 2020*, pages 440–462. Springer, 2020.

[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.

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

[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.

[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.

[MTPBH] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *Proceedings on Privacy Enhancing Technologies*, 2021(4):291–311.

[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.

[Ope22] OpenFHE. Open-Source Fully Homomorphic Encryption Library. `https://github.com/openfheorg/openfhe-development`, 2022.

[PAL21] PALISADE. Lattice Cryptography Library (release 1.11.7). `https://palisade-crypto.org/`, 2021.

[Par21] Jeongeun Park. Homomorphic encryption for multiple users with less communications. *IEEE Access*, 9:135915–135926, 2021.

[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.

[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.