Rotation Key Reduction for Client-Server Systems of Deep Neural Network on Fully Homomorphic Encryption

Joon-Woo Lee[1], Eunsang Lee*[2], Young-Sik Kim*[3], and Jong-Seon No[4]

[1] School of Computer Science and Engineering, Chung-Ang University, Seoul, Republic of Korea
[2] Department of Software, Sejong University, Seoul, Republic of Korea
[3] Department of Electrical Engineering and Computer Science, DGIST, Daegu, Republic of Korea
[4] Department of Electrical and Computer Engineering, INMC, Seoul National University, Seoul, Republic of Korea

Abstract. In this paper, we propose a new concept of *hierarchical rotation key* for homomorphic encryption to reduce the burdens of the clients and the server running on the fully homomorphic encryption schemes such as Cheon-Kim-Kim-Song (CKKS) and Brakerski/Fan-Vercauteran (BFV) schemes. Using this concept, after the client generates and transmits only a small set of rotation keys to the server, the server can generate any required rotation keys from the public key and the smaller set of rotation keys that the client sent. This proposed method significantly reduces the communication cost of the client and the server, and the computation cost of the client. For example, if we implement the standard ResNet-18 network for the ImageNet dataset with the CKKS scheme, the server requires 617 rotation keys. It takes 145.1s for the client with a personal computer to generate whole rotation keys and the total size is 115.7GB. If we use the proposed two-level hierarchical rotation key system, the size of the rotation key set generated and transmitted by the client can be reduced from 115.7GB to 2.91GB ($\times 1/39.8$), and the client-side rotation key generation runtime is reduced from 145.1s to 3.74s ($\times 38.8$ faster) without any changes in any homomorphic operations to the ciphertexts. If we use the three-level hierarchical rotation key system, the size of the rotation key set generated and transmitted by the client can be further reduced from 1.54GB ($\times 1/75.1$), and the client-side rotation key generation runtime is further reduced to 1.93s ($\times 75.2$ faster) with a slight increase in the key-switching operation to the ciphertexts and further computation in the offline phase.

Keywords: Brakerski/Fan-Vercauteran (BFV) schemes · Cheon-Kim-Kim-Song (CKKS) schemes · Fully homomorphic encryption · Hierarchical rotation key · Privacy-preserving machine learning

* Eunsang Lee and Young-Sik Kim are co-corresponding authors.

1 Introduction

Fully homomorphic encryption (FHE) is an encryption scheme which supports the evaluation of arbitrary boolean or arithmetic operations on encrypted data. It is a primary solution for the privacy issue of outsourcing computation, which enables the clients to securely entrust enterprises to process their private information while preserving privacy. The main application includes machine learning [15, 26, 27], genomic analysis [6, 23, 24], cloud services [25], and AI-as-a-service (AIaaS) [31]. Especially, the privacy-preserving AIaaS system is deemed to be one of the most promising techniques, where the clients provide the encrypted data on the cloud and the server processes the data by using the deep neural network, while preserving the privacy of clients' data. Thus, data privacy via FHE is getting more important.

Among various FHE schemes, Cheon-Kim-Kim-Song (CKKS) [9, 11] and Brakerski/Fan-Vercauteran (BFV) [5, 14] schemes are two of the most practical FHE schemes. They can support arithmetic operations for complex numbers or integers in the single-instruction multiple-data (SIMD) manner. Thus, several data can be encrypted in one ciphertext, and one homomorphic operation can simultaneously perform component-wise operations on these multiple message data. Since the CKKS scheme deals with real or complex number data and supports approximate computation on the encrypted real or complex data, it fits the situation allowing approximate computation. On the other hand, the BFV scheme deals with integer data and supports exact computation on the encrypted integer data, and it fits the situation requiring exact computation. The CKKS and BFV schemes also support rotation operation, corresponding to a cyclic shift of message data within a ciphertext. Specifically, it means the cyclic shift operations for the encrypted message vector in that of the CKKS scheme and for rows of the encrypted matrix in one ciphertext of the BFV scheme. The homomorphic rotation operation is inevitable if we require operations between data at different locations in one ciphertext, such as the bootstrapping [3,7,8,28,29], the matrix multiplication [20], and the convolution in convolutional neural networks [22, 26, 27].

In order for the server to perform a rotation operation, an evaluation key for the operation is required, called a *rotation key*. In the client-server model, the client owns the data but wants to delegate the data processing operations to the server, and the server handles the client's data with abundant computational resources instead of the client. Therefore, a secret key capable of decrypting the encrypted data is privately owned by the client. The server should ask the client for a rotation key because the secret key is required to generate these rotation keys. Then the client generates rotation keys with the secret key and sends them to the server. It is similar to generating a public key from a private key in a public key encryption system. (In this paper, the public key only refers to the key used for encryption.) If the rotation of several types of cyclic shifts is required in the homomorphic computation, a distinct rotation key is required for each cyclic shift. Therefore, the server should identify all the cyclic shifts of rotation operation in its computation model in advance and request the corresponding

rotation keys to the client. However, this causes several serious problems when trying to use FHE in the industry, as in the following subsection.

1.1 Rotation Key Problems

Communication costs for rotation keys In the case of complex systems such as machine learning systems, there are many cyclic shifts required for servers to perform the computation model homomorphically. Therefore, the number of the rotation keys that the client has to transmit becomes very large, and the amount of communication that the client and the server should bear dramatically increases. For example, assume that we implement the standard ResNet-20 network for the CIFAR-10 dataset with pre-trained parameters on the CKKS scheme with the polynomial modulus degree $N = 2^{16}$ using the techniques in [26]. Then the size of all 32767 $(= 2^{15} - 1)$ rotation keys is about 13TB, which is an extremely large size to be sent from the client. If the server requests only the required rotation keys for the ResNet-20 network, the server requires 265 rotation keys, corresponding to transmission of 105.6GB. If we design the ResNet-18 network for the ImageNet dataset using the same techniques, 617 rotation keys are required, and it occupies 197.6GB of memory in the server.

Computational costs in the client The computational cost for generating whole rotation keys is also a huge burden on the client. Since the client is assumed not to have high-performance computing devices, the runtime for generating these huge amounts of rotation keys can be very large. For example, even if we have a computer with an AMD Ryzen Threadripper PRO 3995WX CPU processor, a high-performance CPU, it takes 13 minutes to generate whole rotation keys, which is too long to wait for generating keys. Thus, it is desirable to reduce the runtime for rotation key generation in the client.

Lack of flexibility for various services The server may need to support various services for the client. In this case, the server should request a distinct rotation key set required for each service to the client. Whenever a new computation model is added or an existing model is modified in the server, a new rotation key set should be requested. It is a serious problem for the client because the client has to send a new rotation key set whenever the server's computational model is improved and updated. Someone may think that the server simply needs to receive all kinds of rotation keys in advance because of the uncertainty of the model. However, for CKKS schemes using $N = 2^{16}$, this solution is virtually impossible because generating all kinds of rotation keys requires the transmission and storage of 13 TB of rotation keys.

Information leakage of computation model The server should find what kind of cyclic shifts is used in the computation model and request the corresponding rotation keys to the client. However, based on the type of cyclic shifts

required by the server, the client can reasonably infer some information about the computation model of the server. These computational models are important secret assets of the server, and thus they usually do not want to leak any information about the computational model to the client. To solve this situation, the server may deliberately request additional unnecessary rotation keys to confuse the client so that it does not infer, but it causes additional communication and computation costs by increasing the amount of rotation keys to be requested.

Inefficient memory management Since the server usually handles a large number of clients, a large amount of memory is required to store their rotation keys. For example, if a server handles 10,000 clients and requires 500GB of memory per client to store rotation keys for several specific services, 5000TB of memory is required only to store their rotation keys. If each client uses the service infrequently, the server may want to reduce the memory share of the rotation key by temporarily removing the rotation keys that are not currently in use and generating them again when necessary. However, once the rotation keys are removed, the server should request the client to generate and transmit them again.

1.2 Our Contributions

In the conventional rotation key system in CKKS and BFV schemes, all rotation keys should have been generated only through secret keys. However, in the proposed new hierarchical rotation key system, all rotation keys can be generated from public keys or other rotation keys without generating them from secret keys. Specifically, the client creates a small number of so-called "master rotation keys" and sends them to the server with the public key. The server can then generate all required rotation keys from the public key by using the "master rotation keys." That is, the server may convert the public key into a rotation key corresponding to an arbitrary cyclic shift.

We find that the "master rotation key," which allows servers to convert public keys into rotation keys, can be designed by placing a hierarchy on the rotation keys. Therefore, we name the proposed rotation key system the *hierarchical rotation key system*. Rotation keys in the higher level can be used to generate rotation keys in the lower level. The previously mentioned "master rotation key" corresponds to a rotation key in the highest level, and the rotation keys for services are rotation keys in the lowest (level-0) level. These levels are divided according to the size of the total modulus of each rotation key and the special modulus values used.

We propose two fundamental key generation algorithms in the hierarchical rotation key system. The first is a $\mathsf{PubToRot}$ algorithm that generates a low-level r-shift rotation key from the public key by using the high-level r-shift rotation key. The second is a $\mathsf{RotToRot}$ algorithm that generates a low-level $r + r'$-shift rotation key from a low-level r'-shift rotation key by using a high-level r-shift rotation key. If the client generates and sends a highest-level rotation

key set corresponding to cyclic shift set $S = \{r_0, r_1, \cdots, r_{t-1}\}$ to the server, the server can generate the required lower-level $\sum_{i=0}^{t-1} w_i r_i$-shift rotation keys for some non-negative integers w_i's from the public key by sequentially computing PubToRot operations and RotToRot operations. If the client generates and sends the highest-level rotation keys for cyclic shifts of the powers of some integer p, the server can generate any rotation keys by using p-ary number system. If the server needs to generate a set of many low-level rotation keys simultaneously, it can make the most existing low-level rotation keys to generate each low-level rotation key with minimal operations.

The key idea for the proposed hierarchical rotation key system is that the rotation keys can be generated by using rotation operation to the other rotation keys or the public key. In the proposed hierarchical rotation key system, a public key is treated as a single ciphertext and each rotation key is treated as a set of ciphertexts. If we perform the rotation operation to each ciphertext in a rotation key, a new rotation key for other cyclic shifts can be derived. Since the rotation operation requires a rotation key with a larger modulus than the ciphertext, the rotation key for this rotation key generation should have a higher level than the newly generated rotation key. Thus, we propose to set some hierarchy in the rotation keys by the modulus size. We define the rotation key with a larger modulus as the rotation in 'higher key level'. Thus, each rotation key can generate rotation keys with a lower key level. The client-server system using the hierarchical rotation key system has the following improvements compared to the conventional client-server system using FHE schemes, which solves the above five key problems completely.

i) The communication cost between the client and the server is significantly reduced because the clients only need to transmit a small set of high-level rotation keys.
ii) The computation cost of the client is reduced since the clients can generate only a small set of rotation keys.
iii) Even if multiple services need to be requested by the client or the computation model changes in the middle of the services, the server can create the additional rotation keys without additional requests to the client, making the service more flexible.
iv) Since the server can generate the rotation keys with cyclic shifts required for its computation model, it does not have to disclose this information to the client, preventing information leakage about the computation model.
v) The server can temporarily remove unused rotation keys and regenerate them through high-level rotation keys when needed, thereby significantly lowering the overall memory share of the rotation keys.

Fig. 1 shows an example using a hierarchical rotation key system. Fig. 1(a) illustrates a case when an FHE-based service is provided with a conventional method. If the server can provide various types of services for each client, the client must generate and transmit all the rotation keys required for each service to the server. Therefore, when there are many services that can be provided, the

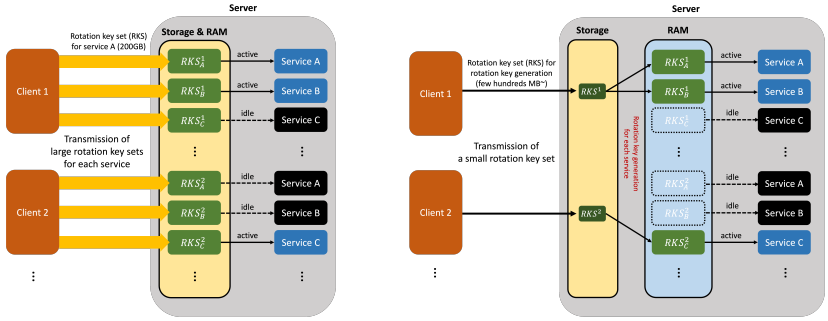
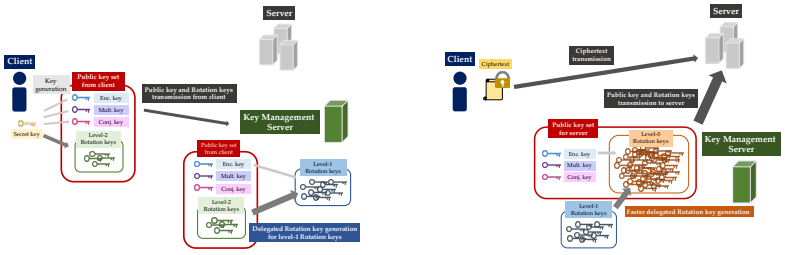


Fig. 1: Comparison of conventional rotation key system and hierarchical rotation key system for HE-based services in the client-server model (a) Conventional rotation key system (b) The proposed hierarchical rotation key system.

burden on the client increases. Even if the client is currently using only a specific service, there is a burden that the server must store all the rotation keys corresponding to all services that the client is not using. Fig. 1(b) shows a case when a hierarchical rotation key system is used. Regardless of how many services are used or how complex the operation is required, the client only needs to send the few high-level rotation keys that the server needs to create rotation keys. In the case of homomorphic encryption parameters that support bootstrapping, communication amount can be reduced from a few hundred GB to several hundred MB. Thereafter, the server can generate rotation keys suitable for the service to be requested by the client and use them for the service. In addition, in the case of a service that the client does not use immediately, the corresponding rotation keys may be removed to use the memory efficiently. If the client wants to resume these services, it can generate and send rotation keys directly with high-level rotation keys to provide the service.

For the contribution 2, some readers may think that the computation amount of the client has simply shifted to the server, and there is no improvement in terms of computation amount. However, due to the nature of the client-server model, which is appropriate for FHE, the server can use high-performance machines and is ready for abundant operations, but the client is not supposed to have high-performance machines and wants to do minimal operations. For this reason, it is a desirable direction in the client-server model that the burden of clients is reduced by transferring a lot of the computational burden of clients to servers, and recent studies related to HE focus on this direction [12, 16].

If the hierarchies of rotation keys are further subdivided, various trade-offs can be adjusted. For example, the fewer high-level rotation keys a client creates, the more time the server will need to generate the rotation key. But if there is a hierarchy with three types of key levels, we can mitigate this trade-off.



(a) (b)

Fig. 2: Efficient rotation key management in three-level hierarchical rotation key generation. (a) Public key and level-2 rotation key transmission from the client and preparation for faster level-0 rotation key generation by generating level-1 rotation keys in advance. (b) Faster rotation key generation from public key and level-1 rotation keys.

Assume that there is an interval between the time when the rotation keys are transmitted and the time when the service is provided. Then, as in Fig. 2, the server can generate a more intermediate level of rotation keys after receiving the highest level of rotation keys. When the server prepares to provide the service (Fig. 2(a)), it is possible to use the highest-level rotation keys and the middle-level rotation keys to generate the required (lowest-level) rotation keys for the service more quickly (Fig. 2(b)). To increase the degree of freedom in rotation key management in this way, we propose a generalized hierarchical rotation key system at multiple key levels.

We further propose several optimization methods for servers to generate keys more efficiently. In most cases, the server must generate a bundle of rotation keys required for a specific service at once. When multiple rotation keys are generated at once, there are many situations in which several types of rotation keys should be generated from one rotation key. In this situation, a *hoisting technique* that can be processed by merging some of the key generation operations is specifically proposed. This reduces the amount of computation by about half. In addition, the amount of computation varies greatly depending on the order of rotation key generation. Finding an efficient generating order for optimization of computation amount can be reduced to a minimum spanning tree problem, which can be solved using Prim's algorithm or Edmonds' algorithm.

We conduct the simulation with the proposed rotation key generation system for the ResNet-20 model with the CIFAR-10 dataset and ResNet-18 model with the ImageNet dataset using an appropriate computing environment for the client-server model. If we implement the standard ResNet-20 network for the CIFAR-10 dataset and the ResNet-18 network for the ImageNet dataset with the CKKS scheme, the server requires 265 and 617 rotation keys. It takes 88.7s and 145.1s for the client with a personal computer to generate whole rotation keys and the total size is 38.3GB and 115.7GB, respectively.

If we use the proposed two-level hierarchical rotation key system to the ResNet-20 model and the ResNet-18 model, the size of the rotation keys generated and transmitted by the client can be reduced from 38.3GB to 8.31GB ($\times 1/4.61$) and from 115.7GB to 2.91GB ($\times 1/39.8$), respectively. The client-side rotation key generation runtime is reduced from 88.7s to 18.9s ($\times 4.69$ faster) and from 145.1s to 3.74s ($\times 38.8$ faster) without changing any homomorphic operations to the ciphertexts. The server-side rotation key generation requires 16.6s and 12.5s. If we may need to make the rotation key generation more efficient at the expense of slightly increasing the key-switching operation time for the ResNet-20 model and the ResNet-18 model, the communication amount for rotation keys significantly reduced to 2.67GB ($\times 1/14.3$) and 1.97GB ($\times 1/58.7$), respectively. The client-side rotation key generation runtime is reduced to 6.30s ($\times 14.1$ faster) and 2.52s ($\times 57.6$ faster), respectively. The server-side rotation key generation requires 10.2s and 16.1s. If we use the three-level hierarchical rotation key system for the ResNet-18 model, the size of the rotation key set generated and transmitted by the client can be further reduced to 1.54GB ($\times 1/75.1$), and the client-side rotation key generation runtime is further reduced to 1.93s ($\times 1/75.2$) with a slight increase in the key-switching operation to the ciphertexts and further computation in the offline phase.

Thus, our contribution can be summarized as follows.

- We point out that the rotation key problem of homomorphic ciphers causes many practical problems in the client-server model and propose a concept of a hierarchical rotation key system that can solve these problems.
- To implement a hierarchical rotation key system, fundamental algorithms that can convert the public key into rotation keys were designed to allow the server to generate arbitrary rotation keys from the public key.
- We propose optimization methods that can effectively reduce the computation amount of the server when it needs to generate many kinds of rotation keys.
- We conduct the simulations showing that we can reduce the computational and communication volumes of clients needed for the machine learning services with the ResNet models for the CIFAR-10 or the ImageNet by $\times 4 \sim \times 75$.

1.3 Related Works

Benes Network and Composition of Rotation Operations Halevi and Shoup [17] suggested a method for any permutation on ciphertext with only $\log n$ types of rotation cyclic shift, where n is the number of slots. It uses the fact that all permutations can be represented as a weighted sum of the $\log n$ number of shifted vectors with fixed different cyclic shifts. If the server has only $\log n$ types of rotation keys for different cyclic shifts, all permutations on ciphertext can be performed with the weighted sum of the $\log n$ number of ciphertexts performed by rotation operation with different corresponding cyclic shifts. Thus, if we want to rotate the ciphertext with a specific cyclic shift that

is different from the designated cyclic shift from the rotation keys the server has, we can deal with this rotation as a type of permutation so that this can also be performed with the above operation.

However, this simple technique has a crucial drawback in the latency for homomorphic computation. Since the rotation operation is one of the most time-consuming homomorphic operations among the elementary homomorphic operations, the number of rotation operations is very sensitive in that it has a direct effect on the whole latency. If there are many required types of cyclic shift for the rotation operation in the homomorphic computation and the server has whole corresponding types of rotation keys, the server may perform only one rotation operation for each cyclic shift. But if the server has only $\log n$ types of rotation keys with the same situation, the server should perform several rotation operations on average for each cyclic shift. For the simple experiment for the ResNet-20 network model in [26] performed with CPU, the latency of this model becomes 1471.2s with Benes network while the latency of this model is 646.8s without Benes, which is a $\times 2.27$ slower result. As the reduction of the long runtime in homomorphic operation is the most sensitive issue in PPML on HE, the simple application of technique in [17] is not desirable. This point is actually the key motivation for this proposal of the hierarchical rotation key concepts.

Our proposed technique solves this issue, in that the clients may generate and send only $\log n$ rotation keys rather than all required rotation keys, and the server can perform only one rotation operation for any cyclic shifts as the server generates any required rotation keys by itself. Since any previous techniques cannot replace this proposed technique for this purpose, the proposed technique is novel in the PPML on HE research area.

Similarly, there are many other techniques in which a slot vector in a ciphertext is rotated or permuted only with a fixed set of rotation keys. Note that the proposed technique does not correspond to this situation but is for the situation that each rotation operation for the ciphertext has to be performed with only one corresponding rotation key because of the efficient latency for ciphertext processing. We perform the key-switching operation with a fixed set of high-level rotation keys to the "rotation keys", not to the "ciphertexts", which are essentially different techniques.

Transciphering technique The aim of this work is very similar to the works for the *transciphering technique* [12, 16]. These two studies aimed to reduce the computational burden and communication burden of the client by making the server bear the computational load of the client. These studies focused on the client's encryption process and the transmission of ciphertexts. In the case of FHE, it is burdensome for clients to encrypt their data and transmit ciphertexts because the amount of computation used in the encryption process is larger and the size of the ciphertext is larger than that of the symmetric key encryption system. Therefore, these studies proposed algorithms to convert symmetric key ciphertexts into ciphertexts in FHE without a secret key. If the client encrypts

data through the encryption process of the symmetric key encryption system and sends it to the server, the server can convert the symmetric key ciphertext into the ciphertext in FHE, reducing the client's computation and communication burden. These studies also mainly mention that the computational power of servers is overwhelmingly stronger than that of clients.

Our paper is the first to note that the same problem is prominent in the rotation keys in homomorphic encryption and to present a solution. In particular, when implementing a privacy-preserving machine learning system using FHE, the size of the encrypted image is hundreds of MB, but the size of the rotation keys required to perform one advanced machine learning system is hundreds of GB. Therefore, in order to apply the machine learning system using FHE to the industry, it is much more important to study the solution to a similar problem as in the transciphering technique. In this respect, our work is consistent with recent research directions in designing FHE systems suitable for client-server systems.

Privacy-preserving deep neural networks on FHE CNN models were implemented using only leveled HE that does not use bootstrapping. The types and numbers of operations available on these CNNs were very limited. The operations used in the model were forced to be modified to be easy to compute on FHE, and the number of layers used was also very limited. Recently, the results of successfully implementing the deep neural network on FHE by actively utilizing bootstrapping operations have been presented [26, 27]. In these studies, the ResNet model, a renowned CNN model with verified classification performance in plaintext, was performed on FHE using the same pre-trained parameters. In particular, in the study of Lee et al. [26], a ResNet model with 110 layers was also successfully implemented on the CKKS scheme.

We find that the massive size of the rotation key is a major obstacle to making this privacy-preserving machine learning system practical when implementing such a complex advanced machine learning system with FHE. Previously, the size of the rotation keys was not a problem because only simple computation models were used. However, bootstrapping operations and various kinds of convolution operations were used in the advanced computation model, which greatly increased the size of the rotation keys to hundreds of GB. Our work can be seen as the study that solves the most significant problem at this point in time when the study of performing advanced computational models with FHE began.

1.4 Outline

Section 2 formalizes the concept of the hierarchical rotation key system and its application to the specific rotation key management. Section 3 deals with the proposed hierarchical rotation key generation algorithm for CKKS and BFV schemes. Section 4 proposes algorithms for efficiently generating a set of rotation keys with the given set of rotation keys in the higher levels. Section 5 suggests

a concrete example of the rotation key management protocol and shows the numerical simulation results with ResNet models. Section 6 concludes the paper and suggests future work.[5]

2 Concept of Hierarchical Rotation Key System

In this section, we provide an overview of the proposed hierarchical rotation key system. Specific procedures in this system will be described in Sections 3 and 4.

2.1 Definition of Hierarchical Rotation Key System

We define the hierarchical rotation key system in the cloud computing using FHE. In a k-level hierarchical rotation key system, there are k sets of rotation keys with a hierarchy from a key level $k-1$ to 0. Each rotation key can be used to generate rotation keys in the lower levels. The additional algorithms for the hierarchical rotation key system are $\mathsf{InitRotKeyGen}$ and $\mathsf{RotKeyGen}$. The algorithm $\mathsf{InitRotKeyGen}$ generates a set of the highest key-level rotation keys, performed by the client with the secret key. The algorithm $\mathsf{RotKeyGen}$ generates a set of intermediate or zero key-level rotation keys using the public key and the set of higher key-level rotation keys. This algorithm is performed by the server or the key management server (KMS) having no secret key. We now assume that the public key and hierarchical rotation keys are managed by the KMS collocated with or separated from the server, and all protocols in the paper also make sense when the KMS and the server are united. These two algorithms are defined as follows, where k denotes the total number of key levels.

- $\mathsf{InitRotKeyGen}(s, \mathcal{T}_{k-1}) \to \{gk_i^{(k-1)}\}_{i\in\mathcal{T}_{k-1}}$: Given a secret key s and a set of cyclic shifts $\mathcal{T}_{k-1}$, generate the rotation keys with cyclic shifts in $\mathcal{T}_{k-1}$ in the highest key level in the client.
- $\mathsf{RotKeyGen}(\ell, \mathcal{U}_\ell, \{gk_i^{(\ell_i)}\}_{i\in\mathcal{U}_\ell}, pk, \mathcal{T}_\ell) \to \{gk_i^{(\ell)}\}_{i\in\mathcal{T}_\ell}$: Given a public key pk, a set of the rotation keys $\{gk_i^{(\ell_i)}\}_{i\in\mathcal{U}_\ell}$ with cyclic shifts in $\mathcal{U}_\ell$ in the key level ℓ_i higher that ℓ, and a set of cyclic shifts $\mathcal{T}_\ell$, generate the rotation keys with cyclic shifts in $\mathcal{T}_\ell$ in key level ℓ in the KMS.

The rotation key $gk_i^{(\ell)}$ denotes the rotation key for the cyclic shift i in the key level ℓ, whose specific definition will be dealt with in Section 3. Although the public key pk is represented separately from the rotation keys, the rotation keys are also public in that these keys can be opened to the public. The set of cyclic shifts for each key level, which is an integer set, is denoted by $\mathcal{T}_0, \cdots, \mathcal{T}_{k-1}$, respectively. These sets are pairwisely disjoint. The set of cyclic shifts for each key level higher than ℓ whose rotation keys are generated in advance, is denoted by $\mathcal{U}_\ell$. If all desired rotation keys in the key level higher than ℓ are all generated,

[5] The appendix includes the preliminaries, the proofs of the theorems, and the required cyclic shifts for ResNet models.

Algorithm 1: Key Management of Hierarchical Rotation Key System with the k Key Levels

Input: Encryption parameters $params$ for k-level rotation key system (client and server), a set of cyclic shifts for rotation keys in the highest key level $\mathcal{T}_{k-1}$ (client), sets of cyclic shifts for intermediate key-level rotation keys $\mathcal{T}_{k-2}, \cdots, \mathcal{T}_1$ (server), and a homomorphic service $\mathcal{S}$ (server)

Output: A set of rotation keys $\{gk_i^{(0)}\}_{i\in\mathcal{T}_0}$ (server)

Key generation and transmission in client

1. $sk \leftarrow \mathsf{SecGen}(1^\lambda, params)$
2. $pk \leftarrow \mathsf{PubGen}(sk)$
3. $\{gk_i^{(k-1)}\}_{i\in\mathcal{T}_{k-1}} \leftarrow \mathsf{InitRotKeyGen}(s, \mathcal{T}_{k-1})$
4. Transmit $(pk, \{gk_i^{(k-1)}\}_{i\in\mathcal{T}_{k-1}})$ to the server and let $\mathcal{G} = \{gk_i^{(k-1)}\}_{i\in\mathcal{T}_{k-1}}$

Inactive phase: generating rotation keys in the key level $\ell(>1)$

1. $\{gk_i^{(\ell)}\}_{i\in\mathcal{T}_\ell} \leftarrow \mathsf{RotKeyGen}(\ell, \mathcal{U}_\ell, \{gk_i^{(\ell_i)}\}_{i\in\mathcal{U}_\ell}, pk, \mathcal{T}_\ell)$
2. $\mathcal{G} \leftarrow \mathcal{G} \cup \{gk_i^{(\ell)} : i \in \mathcal{T}_\ell\}$

Active phase: generating level-0 rotation keys in server

1. $\mathcal{T}_0 \leftarrow \mathsf{ExtractRotSet}(\mathcal{S})$
2. $\{gk_i^{(0)}\}_{i\in\mathcal{T}_0} \leftarrow \mathsf{RotKeyGen}(0, \mathcal{U}_\ell, \{gk_i^{(1)}\}_{i\in\mathcal{T}_\ell}, pk, \mathcal{T}_0)$

$\mathcal{U}_\ell$ is equal to $\bigcup_{i=\ell+1}^{k-1} \mathcal{T}_i$. The conventional rotation key system can be seen as a special case of the proposed hierarchical rotation key system, where there is only the algorithm $\mathsf{InitRotKeyGen}$, and the number of key levels in the hierarchy is one.

2.2 Rotation Key Generation Protocol in Hierarchical Rotation Key System

In our k-level hierarchical rotation key system, we consider both of the following cases: (1) *active* case when a client frequently requests a service and (2) *inactive* case when a service is not used often by a client or what service to be used is not determined. In the active case, since the generation of level-0 rotation keys repetitively for each service is inefficient, it is better to generate level-0 rotation keys in advance to reduce latency. In this case, the server should hold level-0 rotation keys, which takes up some memory. On the other hand, in the case of inactive, it is desirable to generate only rotation keys of level 1 to $k-1$ in advance to remove unnecessary memory usage of level-0 rotation keys. In this way, our system can finely control the trade-off of memory and latency according to the frequency of service. The specific protocols are described in Algorithm 1.

3 Proposed Hierarchical Rotation Key System for CKKS and BFV Schemes

In this section, the hierarchical rotation key system for CKKS and BFV schemes is proposed. The CKKS and BFV schemes differ only in the packing structure, the decryption method, and the role of each operation for the encrypted data, but the key-switching operation itself is exactly the same. Thus, we will deal with them at once.

We use the term *ciphertext* as a pair of ring elements $(b, a) \in R_q^2$ for some modulus q. A ciphertext $(b, a) \in R_q^2$ is defined to be a valid ciphertext of m with the secret key s if $b + a \cdot s = m + e \mod q$, where e is a polynomial with small coefficients compared to q.

Let $Q = \prod_{i=0}^{\mathtt{dnum}-1} Q_i$ be a product of several coprime positive integers Q_i's, and P be a positive integer which is coprime to and larger than Q_i's. A *rotation key* $\mathsf{gk}_r = \{\mathsf{gk}_{r,i}\}_{i=0,\cdots,\mathtt{dnum}-1}$ for cyclic shift r with the secret key polynomial $s \in R$ is defined to be valid if each $\mathsf{gk}_{r,i} = (b_{r,i}, a_{r,i}) \in R_{PQ}^2$ is a valid ciphertext of $P \cdot \hat{Q}_i \cdot [\hat{Q}_i^{-1}]_{Q_i} \cdot s(X^{5^r})$ with the secret key s, where $\hat{Q}_i = \prod_{j \neq i} Q_j$. This can be used for the key-switching operation to the ciphertext in the modulus q, where q is a divisor of Q. We call Q the evaluation modulus and P the special modulus.

These rotation keys are used in the rotation operation. The rotation operation of the CKKS scheme is an operation mapping $(v_i) \mapsto (v_{i+r})$ while encrypted, where the addition operation of the subscript is in modulo $N/2$ and N is the polynomial modulus degree. The rotation operation in the BFV scheme is an operation mapping $(v_{i,j}) \mapsto (v_{i,(j+r)})$ while encrypted. In terms of ring elements, these operations can be unified as operations mapping $m(X) \mapsto m(X^{5^r})$. For these operations, we first perform an operation of $(b(X), a(X)) \mapsto (b(X^{5^r}), a(X^{5^r}))$. This processed ciphertext satisfies $b(X^{5^r}) + a(X^{5^r}) \cdot s(X^{5^r}) \approx m(X^{5^r})$, which means that it is a ciphertext of a plaintext polynomial $m(X^{5^r})$ with the secret key $s(X^{5^r})$. We have to convert this ciphertext to a ciphertext of the same plaintext with the original secret key. This is done by taking the key-switching operation from $s(X^{5^r})$ to $s(X)$ using the rotation key for cyclic shift r.

3.1 Generation of Public Key and Rotation Keys in Client

The conventional schemes generate a public key (b, a) with the modulus $Q = \prod_{i=0}^{L} q_i$ because the special modulus is only used in the key-switching operation. In contrast, the proposed hierarchical rotation key generation scheme generates a public key (b, a) with $Q_{k-1} = \prod_{i=0}^{L_{k-2}} q_i$ to prepare to use it to generate rotation keys with key levels smaller than $k - 1$. The highest key-level rotation keys are generated by the client. The set of cyclic shifts $\mathcal{U}_\ell$ of rotation keys in the key level higher than ℓ should be the set that can generate all cyclic shifts $\mathcal{T}_\ell$ of rotation keys with the key level ℓ by the sum allowing repetition. The small size of $\mathcal{T}_{k-1}$ for the highest key level can reduce the computational burden and the communication cost of the client.

In the InitRotKeyGen operation for the proposed scheme, a single highest-level rotation key for cyclic shift r with the secret key polynomial $s \in R$ is the form of $\mathsf{gk}_r^{(k-1)} = \{\mathsf{gk}_{r,i}^{(k-1)}\}_{i=0,\cdots,\mathtt{hdnum}_{k-1}-1}$, where $\mathsf{gk}_{r,i}^{(k-1)} = (b_{r,i}^{(k-1)}, a_{r,i}^{(k-1)}) \in R_{Q_{k-1}P_{k-1}}^2$ such that $a_{r,i}^{(k-1)} \leftarrow R_{Q_{k-1}P_{k-1}}$ and $b_{r,i}^{(k-1)} = -a_{r,i}^{(k-1)} \cdot s + e_{r,i}^{(k-1)} + P_{k-1} \cdot \hat{Q}_{k-1,i} \cdot [\hat{Q}_{k-1,i}^{-1}]_{Q_{k-1,i}} \cdot s(X^{5^r})$ for $e_{r,i}^{(k-1)} \leftarrow \chi$. The RNS bases for $\mathsf{gk}_{r,i}$ are $\mathcal{C} \cup \bigcup_{j=0}^{k-1} \mathcal{B}_j$. Note that the distribution and the form of the rotation keys generated by the client are the same as those in the conventional rotation key generation.

3.2 RotToRot and PubToRot Operations

Two types of operations are required to make the level-ℓ rotation keys for ℓ less than $k-1$. One is the operation PubToRot, which generates a level-ℓ rotation key from the public key, and the other is the operation RotToRot, which generates a level-ℓ rotation key from the existing level-ℓ rotation keys for the other cyclic shifts. The combination of PubToRot operation and RotToRot operation will generate all rotation keys with only the public key and rotation keys in the key level higher than ℓ.

Let the shift-r rotation key be defined as the rotation key for cyclic shift r, and let (r, ℓ) rotation key be defined as the rotation key for cyclic shift r in the key level ℓ. For the convenience of explanation, we will first explain the operation RotToRot. The operation RotToRot is an operation that generates a $(r + r', \ell)$ rotation key from a (r, ℓ) rotation key in the key level ℓ with a shift-r' rotation key in the key level higher than ℓ. To understand this operation, keep in mind that the rotation operation is a map $m(X) \mapsto m(X^{5^r})$ from the perspective of the plaintext polynomial. In other words, the rotation operation can be seen as an operation that generates a ciphertext of $m(X^{5^r})$ from a ciphertext of $m(X)$ [11]. We note that the rotation key for cyclic shift r is a set of ciphertexts $\mathsf{gk}_r^{(\ell)} = \{\mathsf{gk}_{r,i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1}$, where $\mathsf{gk}_{r,i}^{(\ell)} = (b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)}) \in R_{Q_\ell P_\ell}^2$ and $b_{r,i}^{(\ell)} = -a_{r,i}^{(\ell)} \cdot s + e_{r,i}^{(\ell)} + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}} \cdot s(X^{5^r})$. Each $\mathsf{gk}_{r,i}^{(\ell)}$ is a ciphertext of $P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}} \cdot s(X^{5^r})$. If we perform the rotation operation with cyclic shift r' on $\mathsf{gk}_{r,i}^{(\ell)}$, the output is a ciphertext of the following polynomial,

$$\begin{aligned} & P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}} \cdot s((X^{5^{r'}})^{5^r}) \\ = & P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}} \cdot s(X^{5^{r+r'}}). \end{aligned}$$

This rotation operation requires an (r', ℓ') rotation key $\mathsf{gk}_{r'}^{(\ell')}$, where ℓ' is higher than ℓ. If we define this output as $\mathsf{gk}_{r+r',i}^{(\ell)}$, the set $\mathsf{gk}_{r+r'}^{(\ell)} = \{\mathsf{gk}_{r+r',i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1}$ is a valid $(r + r', \ell)$ rotation key. We will call this operation RotToRot, as shown in Algorithm 2. The following theorem shows the correctness of RotToRot operation.[6]

[6] The proof of the theorem is in Appendix B.

Theorem 1. *The output of Algorithm 2 is a valid rotation key for the rotation operation for cyclic shift $r + r'$.*

Next, we will describe the operation PubToRot. Note that the above operation is useful only when some rotation keys exist. However, since the server receives no rotation key in the key level lower than $k-1$ from the client, the rotation key should be generated first with the public key and rotation keys in the higher levels in the server. To this end, we can think of a formal shift-0 rotation key. If a shift-0 rotation key can be generated from a public key, a shift-r' rotation key can be generated by adding a RotToRot operation to the shift-0 rotation key for cyclic shift r'. By definition, the shift-0 rotation key should be the form of $\mathsf{gk}_0^{(\ell)} = \{\mathsf{gk}_{0,i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1}$, where $\mathsf{gk}_{0,i}^{(\ell)} = (b_{0,i}^{(\ell)}, a_{0,i}^{(\ell)}) \in R^2_{Q_\ell P_\ell}$ and $b_{0,i}^{(\ell)} = -a_{0,i}^{(\ell)} \cdot s + e_{0,i}^{(\ell)} + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}} \cdot s$.

To generate $\mathsf{gk}_{0,i}^{(\ell)}$ from the public key $(b, a) \in R^2_{Q_{k-1}}$, we first reduce the public key to $(b', a') = (b \mod Q_\ell P_\ell, a \mod Q_\ell P_\ell) \in R^2_{Q_\ell P_\ell}$ by simply extracting values for corresponding RNS moduli. Then, we set as $b_{0,i}^{(\ell)} = b'$ and $a_{0,i}^{(\ell)} = a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}$. Then, we can have $b_{0,i}^{(\ell)} = -a_{0,i}^{(\ell)} \cdot s + e_{0,i}^{(\ell)} + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}} \cdot s$. If we define $(b_{0,i}^{(\ell)}, a_{0,i}^{(\ell)})$ as $\mathsf{gk}_{0,i}^{(\ell)}$, the set $\mathsf{gk}_0^{(\ell)} = \{\mathsf{gk}_{0,i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1}$ is a valid formal $(0, \ell)$ rotation key. Then we can generate a shift-r rotation key by performing a RotToRot operation on it with the (r, ℓ) rotation key.

In addition, we can optimize the operations further by combining the decomposition processes in the key-switching operation. Trivially, the decomposition process is performed $\mathtt{hdnum}_\ell$ times if all the key-switching operations are performed in a black-box manner like RotToRot. Since the decomposition process is the heaviest operation in the key-switching operation [3], reducing the number of these processes is desirable. Rather than performing the decomposition process after adding $P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}$ to a' for each i, we perform the decomposition process to a' only once and add $[P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}$ to the j-th decomposed component for each i, where this added value can be pre-computed. Since the number of the decomposition processes is reduced to one, this optimization effectively improves the running time performance. The PubToRot operation is shown in Algorithm 3. The correctness of this optimization is shown in the following theorem,[7] where $P_\ell Q_\ell = Q_{\ell+1} = (\prod_{j=0}^{\mu-2} Q_{\ell',j}) \cdot \bar{Q}_{\ell',\mu-1}$, $\bar{Q}_{\ell',\mu-1}$ is a divisor of $Q_{\ell',\mu-1}$, and $\mu \leq \mathtt{hdnum}_{\ell'}$.

Theorem 2. *The output of Algorithm 3 is a valid rotation key for the rotation operation for cyclic shift r.*

3.3 Rotation Key Generation in the Lower Key Level

We can generate the desired level-ℓ rotation keys with only the public key and the rotation keys in the key level higher than ℓ through RotToRot and PubToRot

[7] The proof of the theorem is in Appendix C.

Algorithm 2: RotToRot

Input: An (r, ℓ) rotation key, $\mathsf{gk}_r^{(\ell)} = \{(b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)})\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1} \in (R_{Q_\ell P_\ell}^2)^{\mathtt{hdnum}_\ell}$ and an (r', ℓ') rotation key, where ℓ' is higher than ℓ, $\mathsf{gk}_{r'}^{(\ell')} = \{(b_{r',i}^{(\ell')}, a_{r',i}^{(\ell')})\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1} \in (R_{Q_{\ell'} P_{\ell'}}^2)^{\mathtt{hdnum}_{\ell'}}$

Output: An $(r + r', \ell)$ rotation key, $\mathsf{gk}_{r+r'}^{(\ell)} = \{b_{r+r',i}^{(\ell)}, a_{r+r',i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1} \in (R_{Q_\ell P_\ell}^2)^{\mathtt{hdnum}_\ell}$

for $i = 0$ **to** $\mathtt{hdnum}_\ell - 1$ **do**
  $(\tilde{b}, \tilde{a}) \leftarrow (b_{r,i}^{(\ell)}(X^{5^{r'}}), a_{r,i}^{(\ell)}(X^{5^{r'}}))$
  $(b_{r+r',i}^{(\ell)}, a_{r+r',i}^{(\ell)}) \leftarrow$ key-switching operation to $(\tilde{b}, \tilde{a})$ with the rotation key $\mathsf{gk}_{r'}^{(\ell')}$.
return $\{(b_{r+r',i}^{(\ell)}, a_{r+r',i}^{(\ell)})\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1}$

described in Algorithm 4. It is assumed that a cyclic shift r of a required rotation key can be represented as $r_0 + \cdots + r_{t-1}$, where each r_i is an element in $\mathcal{U}_\ell$, and we deal with the case when only one level-ℓ rotation key is generated. To generate the (r, ℓ) rotation key, we first perform the operation PubToRot with the shift-r_0 rotation key and the public key. Then we perform a RotToRot operation iteratively with the shift-r_i rotation key and the shift-$\sum_{j=0}^{i-1} r_j$ rotation key to generate a shift-$\sum_{j=0}^{i} r_j$ rotation key for $i = 1, \cdots, t-1$, which outputs the (r, ℓ) rotation key at last. The generation algorithm for one rotation key is described in Algorithm 4.

We usually have to generate a bundle of rotation keys rather than only one rotation key for a specific service. We will deal with the more efficient method for the case when we need to make a set of rotation keys at once in Section 4.

3.4 Security Aspects

One can be concerned that the server may be able to obtain some information about the secret key using the fact that the rotation keys for any cyclic shifts can be generated by the server indefinitely. However, according to the argument often used in the simulation paradigm in cryptography, if any new information can be efficiently obtained from existing information, this new information is considered to tell us nothing beyond the existing information [30]. Thus, even if new rotation keys are generated indefinitely with the proposed algorithms from the rotation keys sent by the client, these new rotation keys do not give the server any new information beyond the public keys and the rotation keys in the highest key level sent by the client.

Thus, we only need to consider the security of the public key and the rotation keys at the highest key level sent by the client. As mentioned in Section 3.1, the generating method for the public key and the highest key-level rotation key by the client is exactly the same as those of the conventional FHE schemes. Just as the conventional FHE schemes are based on the circular security assumption, the

Algorithm 3: PubToRot

Input: A public key $(b, a) \in R^2_{Q_{k-1}}$, a (r, ℓ') rotation key, $\mathsf{gk}_r^{(\ell')} = \{b_{r,j}^{(\ell')}, a_{r,j}^{(\ell')}\}_{j=0,\cdots,\mathtt{hdnum}_{\ell'}-1} \in (R^2_{Q_{\ell'}P_{\ell'}})^{\mathtt{hdnum}_{\ell'}}$, and the key level ℓ

Output: A (r, ℓ) rotation key, $\mathsf{gk}_r^{(\ell)} = \{b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_{\ell}-1} \in (R^2_{Q_{\ell}P_{\ell}})^{\mathtt{hdnum}_{\ell}}$

$(b', a') \leftarrow ([b(X^{5^r})]_{Q_\ell P_\ell}, [a(X^{5^r})]_{Q_\ell P_\ell}) \in R^2_{Q_\ell P_\ell}$

Decompose a' into a vector $(a_0, \cdots, a_{\mu-1}) \in R^{\mu}_{P_{\ell'}Q_{\ell+1}}$, where $a_j = [a']_{Q_{\ell',j}} + Q_{\ell',j} \cdot \tilde{e}_j$ for small $\tilde{e}_j$'s for $0 \le j \le \mu - 2$ and $a_{\mu-1} = [a']_{\bar{Q}_{\ell',\mu-1}} + \bar{Q}_{\ell',\mu-1} \cdot \tilde{e}_{\mu-1}$ for small $\tilde{e}_{\mu-1}$.

for $i = 0$ **to** $\mathtt{hdnum}_\ell - 1$ **do**

  $(\bar{b}, \bar{a}) \leftarrow (0, 0) \in R^2_{P_{\ell'}Q_{\ell'}}$

  for $j \leftarrow 0$ **to** $\mu - 1$ **do**

    if $j = \mu - 1$ **then**

      $(\bar{b}, \bar{a}) \leftarrow (\bar{b}, \bar{a}) + (a_{\mu-1} + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{\bar{Q}_{\ell',\mu-1}}) \cdot ([b_{r,\mu-1}^{(\ell')}]_{P_{\ell'}Q_{\ell+1}}, [a_{r,\mu-1}^{(\ell')}]_{P_{\ell'}Q_{\ell+1}})$

    else

      $(\bar{b}, \bar{a}) \leftarrow (\bar{b}, \bar{a}) + (a_j + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}) \cdot ([b_{r,j}^{(\ell')}]_{P_{\ell'}Q_{\ell+1}}, [a_{r,j}^{(\ell')}]_{P_{\ell'}Q_{\ell+1}})$

  $(b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)}) \leftarrow (\lfloor P_{\ell'}^{-1} \cdot \bar{b} \rceil, \lfloor P_{\ell'}^{-1} \cdot \bar{a} \rceil) \in R^2_{Q_{\ell+1}} = R^2_{P_\ell Q_\ell}$

  $b_{r,i}^{(\ell)} \leftarrow b_{r,i}^{(\ell)} + b'$

return $\{(b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)})\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1}$

proposed hierarchical rotation key generation scheme also requires the circular security assumption. The public key is an element of $R^2_{Q_{k-1}}$, and the highest key-level rotation key is an element of $(R_{Q_{k-1}P_{k-1}})^{2})^{\mathtt{hdnum}_{k-1}}$. Since the main factor that affects the security level is the maximum modulus bit-length of rings, the value of $Q_{k-1}P_{k-1}$ is the main factor for security. For a given polynomial modulus degree N and the secret key Hamming weight h, we can be given the maximum modulus bit length to guarantee the security level λ [3, 10], and the bit-length of $Q_{k-1}P_{k-1}$ should not exceed this bit length.

4 Efficient Generation Method of Rotation Key Set

In the previous section, we dealt with the specific algorithms needed to make a lower key-level rotation key using the higher key-level rotation keys. However, we often require many rotation keys at once, especially for certain services requested by the client. Thus, it is necessary to efficiently generate a set of rotation keys using the higher key-level rotation keys. We need to reduce the number of these RotToRot operations and PubToRot operations to efficiently generate hierarchical rotation keys. Note that there are many intermediate rotation keys in the hierarchical rotation key system. Given a certain fixed set of higher key-level

Algorithm 4: RotKeyGen for One Rotation Key

Input: A public key $(b, a) \in R^2_{Q_{k-1}}$, a set of rotation keys $\mathcal{G}_{\mathcal{U}_\ell} = \{\mathsf{gk}_r^{(\ell_r)} = \{(b_{r,i}^{(\ell_r)}, a_{r,i}^{(\ell_r)})\}_{i=0,\cdots,\mathtt{hdnum}_{\ell_r}-1} \in (R^2_{Q_{\ell_r}P_{\ell_r}})^{\mathtt{hdnum}_{\ell_r}} | r \in \mathcal{U}_\ell\}$ for cyclic shift generator set $\mathcal{U}_\ell$ in the key level higher than ℓ, and a cyclic shift $r = \sum_{u=0}^{t-1} r_u$ for $r_u \in \mathcal{U}_\ell$

Output: An (r, ℓ) rotation key,
$\mathsf{gk}_r^{(\ell)} = \{b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell-1} \in (R^2_{Q_\ell P_\ell})^{\mathtt{hdnum}_\ell}$

$\{(b_{r_0,i}^{(\ell)}, a_{r_0,i}^{(\ell)})_{i=0,\cdots,\mathtt{hdnum}_\ell}\} \leftarrow$ PubToRot with the public key and the (r_0, ℓ_{r_0}) rotation key

for $h = 1$ **to** $t-1$ **do**

  $\{(b^{(\ell)}_{\sum_{j=0}^{h} r_j,i}, a^{(\ell)}_{\sum_{j=0}^{h} r_j,i})_{i=0,\cdots,\mathtt{hdnum}_\ell}\} \leftarrow$ RotToRot with $\{(b^{(\ell)}_{\sum_{j=0}^{h-1} r_j,i}, a^{(\ell)}_{\sum_{j=0}^{h-1} r_j,i})_{i=0,\cdots,\mathtt{hdnum}_\ell}\}$ and the (r_h, ℓ_{r_h}) rotation key

return $\mathsf{gk}_r^\ell = \{(b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)})_{i=0,\cdots,\mathtt{hdnum}_\ell}\}$

rotation keys, the key problem is how to minimize the number of operations for generating these intermediate rotation keys by systematically organizing the generating sequence of the rotation keys.

4.1 Reduction to Minimum Spanning Arborescence Problem and Minimum Spanning Tree Problem

Given a set $\mathcal{U}_\ell$ of specific fixed rotation keys in the higher key level than ℓ, generating level-ℓ rotation keys with as few operations as possible is desirable. In other words, it becomes important to use the least amount of operations of RotToRot and PubToRot by arranging the order in which the rotation keys in the set are generated. We propose an algorithm that can determine the order of generating rotation keys in the set in the hierarchical rotation key system to reduce the number of operations.

To this end, we reduce the problem of determining the order of generation of rotation keys to the minimal spanning arborescence problem, a well-known graph-theoretic computational problem. First, set the $|\mathcal{T}_\ell| + 1$ nodes for each element in the $\mathcal{T}_\ell \cup \{0\}$, and then set the directed edge weight between any two nodes a, b as the minimum number of elements in $\mathcal{U}_\ell$ required to add up to $|a-b|$ allowing repetition. The method for setting this edge weight will be given in the next subsection. There are some identical points to the minimum arborescence problem in our ordering of the rotation key generation problem as follows.

- We need to generate each rotation key only once. This fact is related to the property of the arborescence that any node has only one path from the root node.
- Each rotation key can be generated by using a RotToRot or a PubToRot from the public key or existing rotation keys. An edge from the node a to the node

b with weight w means that the (b, ℓ) rotation key can be generated from the (a, ℓ) rotation key with w operations of RotToRot and PubToRot.
- All rotation keys should be generated from the public key and the higher level rotation keys. An arborescence has only one root node that is the source of all nodes, and this root node corresponds to the public key.
- We need to minimize the total number of key-switching operations to generate all rotation keys. The minimum spanning arborescence problem is to find the arborescence with the minimum total weight, which corresponds to the total number of PubToRot operations or RotToRot operations.

Therefore, the graph produced in this way can be seen as a directed graph, and our problem is to find a spanning arborescence with a minimum sum of edges, which is the goal of the minimum spanning arborescence problem. If we find a spanning arborescence in the graph, we can view the node with zero as a public key and generate the rotation keys along the obtained tree. The minimum spanning arborescence problem can be solved by Edmonds' algorithm [13], and thus an answer to this problem can be efficiently obtained.

If the rotation keys in the higher key level exist in pairs of different signs of the same absolute value, a faster and more efficient solution for generating the rotation keys in the lower key level can be obtained by reducing to another computation problem. If a shift-r_1 rotation key can be generated with m operations from a shift-r_2 rotation key, we can generate the shift-r_2 rotation key from that of cyclic shift r_1 with the higher-level rotation keys for cyclic shifts having the same absolute value with a different sign. In view of the corresponding graph, any pairs of two edges (r_1, r_2) and (r_2, r_1) exist and have the same edge weight. Thus, we can replace the directed graph with the undirected graph with the same nodes in which each edge has the same weight as the corresponding edge in the directed graph. For the undirected graph, we can reduce this problem to the minimum spanning tree problem, which can be solved by Prim's algorithm [33].

We note that this solution is not exactly the optimal solution since the insertion of additional nodes can reduce the operations further. If we set the nodes for all cyclic shifts (i.e., $\pm 1, \pm 2, \pm 3, \cdots$) in the graph, our problem is to find the minimum Steiner tree for required cyclic shifts. The Steiner tree in a graph is a tree connecting a subset of designated nodes, and the problem of finding the Steiner tree is known as an NP-hard problem. Thus, we choose the near-optimal solution using a more practically feasible algorithm. Designing a fast algorithm to find the solution closer to the optimal solution in the proposed situation is an important future work.

4.2 Edge Weight for p-ary Rotation Keys

We should consider a method to compute the edge of each graph, where we need to find a way to represent the difference between two nodes as a sum of the minimum number of elements in $\mathcal{U}_\ell$, allowing repetition. In general, the server can ask the client for a well-designed set of $\mathcal{U}_\ell$ so that it can be easy to

represent any given number as the desired sum in $\mathcal{U}_\ell$. Rather than proposing the general method for unstructured $\mathcal{U}_\ell$, we suggest a specific example of a key management system with $\mathcal{U}_\ell$ with power-of-p integers within the desired interval. We will discuss how to obtain edges for both cases when $\mathcal{U}_\ell$ consists of power-of-p integers with both signs and when it consists of only positive power-of-p integers.

Algorithm 5: ComputeEdgePos

Input: A power base p for the set $\mathcal{U}_\ell$ with only positive numbers and a number t to be summed
Output: The minimum number of elements in $\mathcal{U}_\ell$ summed to t allowing repetition

$(t_0 t_1 \cdots t_{\ell-1})_{(p)} \leftarrow p$-ary representation of t
return $\sum_{i=0}^{\ell-1} t_i$

We consider the easier case, a set of positive power-of-p, in which the rotation graph is a directed graph. In this case, each edge can be computed as follows. First, we can find the difference between the end node and the start node of the edge, and then express this difference in the p-ary representation, and then set the sum of the digits as the edge weight. This algorithm is described in Algorithm 5 without proof.

Next, we consider the case of power-of-p integers with both signs in which the rotation graph is an undirected graph, as in Section 4.1. In this case, since the power-of-two integers with different signs can add up to the value, expressing them with the p-ary representation is not enough to find the optimal solution. Instead, we propose the following algorithm to obtain the edge weight between any two nodes, which is efficient enough for the input range. Assume that r is the difference between the two given nodes. If r is a multiple of p, then recursively output a value of $\mathsf{Alg}(r/p)$, otherwise find r_1 such that $pr_1 \leq r < p(r_1+1)$ and recursively output $\min\{\mathsf{Alg}(r_1) + (r - pr_1), \mathsf{Alg}(r_1+1) + (p(r_1+1) - r)\}$. This algorithm is described in Algorithm 6. To help understand the reduction to the graph, we depict the corresponding graph and the minimum spanning tree for $\mathcal{T}_\ell = \{1, 13, 16, 17, 19\}$ and $\mathcal{U}_\ell = \{\pm 1, \pm 2, \pm 4, \pm 8, \pm 16\}$ in Figure 3.

4.3 Hoisted Rotation Key Generation

The previous subsections focus on reducing the number of RotToRot and PubToRot operations. In this subsection, we further reduce the number of the Decompose processes by the hoisting technique. The hoisting technique minimizes the number of operations by interchanging or combining operations without changing functionalities. This technique has been used in the linear transformation in the FHE schemes, and optimizing the bootstrapping of the FHE schemes is one of its important applications [3,18]. We propose the hoisting method for generating the rotation key set in the hierarchical rotation key generation systems.

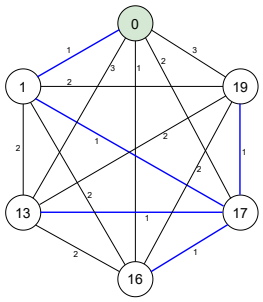


Fig. 3: Rotation key graph for $\mathcal{T}_\ell = \{1, 13, 16, 17, 19\}$ and $\mathcal{U}_\ell = \{\pm 1, \pm 2, \pm 4, \pm 8, \pm 16\}$.

Algorithm 6: ComputeEdgeBoth

Input: A power base p for the set $\mathcal{S}$ with both signs and the number t to be summed

Output: The minimum number of elements in $\mathcal{S}$ summed to t allowing repetition

```
1 if p|t then
2 |  return ComputeEdgeBoth(p, t/p)
3 else
4 |  r ← ⌊|t|/p⌋
5 |  if r = 0 then
6 |  |  return |t|
7 |  else
8 |  |  return min{ComputeEdgeBoth(p, r) + (|t| −
  |  |    pr), ComputeEdgeBoth(p, r + 1) + (p(r + 1) − |t|)}
```

The target situation is when several level-ℓ rotation keys are generated from the public key or one level-ℓ rotation key with rotation keys in the key level ℓ' higher than ℓ. If we want to generate d rotation keys, we can naively perform exactly d PubToRot operations or d RotToRot operations. As stated in Section 3.2, the decomposition process is the most time-consuming in the key-switching operation, and thus the decomposition process is desirable to be reduced further. To this end, we postpone the process of automorphism in line 2 of Algorithm 2 or line 1 of Algorithm 3 to the last of the operations to combine the decomposition processes into one process. To maintain the functionality of the operation, we conduct the automorphism inversely to the rotation keys in the key level higher than ℓ before the inner-product with the decomposed components. If the source rotation key is the public key, we reduce the number of decomposition processes from d to one for generating d rotation keys. If the source rotation key is the other rotation key in the same key level, we reduce the number of decomposition processes from $d \cdot \texttt{hdnum}_\ell$ to $\texttt{hdnum}_\ell$. The hoisted version of RotToRot and PubToRot operations are described in Algorithms 7 and 8. The whole generation

Algorithm 7: HoistedRotToRot

Input: An (r, ℓ) rotation key, $\mathsf{gk}_r^{(\ell)} = \{(b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)})\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1} \in (R_{Q_\ell P_\ell}^2)^{\mathtt{hdnum}_\ell}$ and d (r'_α, ℓ') rotation keys, where ℓ' is higher than ℓ, $\mathsf{gk}_{r'_\alpha}^{(\ell')} = \{(b_{r'_\alpha,i}^{(\ell')}, a_{r'_\alpha,i}^{(\ell')})\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1} \in (R_{Q_{\ell'} P_{\ell'}}^2)^{\mathtt{hdnum}_{\ell'}}$ for $\alpha = 0, \cdots, d-1$

Output: d $(r + r'_\alpha, \ell)$ rotation keys, $\mathsf{gk}_{r+r'_\alpha}^{(\ell)} = \{b_{r+r'_\alpha,i}^{(\ell)}, a_{r+r'_\alpha,i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1} \in (R_{Q_\ell P_\ell}^2)^{\mathtt{hdnum}_\ell}$ for $\alpha = 0, \cdots, d-1$

for $i = 0$ **to** $\mathtt{hdnum}_\ell - 1$ **do**
  Decompose $a_{r,j}^{(\ell)}$ into a vector $(a_0, \cdots, a_{\mu-1}) \in R_{P_{\ell'} Q_{\ell+1}}^{\mu}$, where $a_j = [a]_{Q_{\ell',j}} + Q_{\ell',j} \cdot \tilde{e}_j$ for small $\tilde{e}_j$'s for $0 \le j \le \mu - 2$ and $a_{\mu-1} = [a]_{\bar{Q}_{\ell',\mu-1}} + \bar{Q}_{\ell',\mu-1} \cdot \tilde{e}_{\mu-1}$ for small $\tilde{e}_{\mu-1}$.
  for $\alpha = 0$ **to** $d - 1$ **do**
    $(\bar{b}, \bar{a}) \leftarrow (0, 0) \in R_{P_\ell Q_\ell}^2$
    for $j \leftarrow 0$ **to** $\mu - 1$ **do**
      $(\bar{b}, \bar{a}) \leftarrow (\bar{b}, \bar{a}) + a_j \cdot ([b_{r'_\alpha,j}^{(\ell')}(X^{5^{-r'_\alpha}})]_{P_{\ell'} Q_{\ell+1}}, [a_{r'_\alpha,j}^{(\ell')}(X^{5^{-r'_\alpha}})]_{P_{\ell'} Q_{\ell+1}})$
    $(b_{r+r'_\alpha,i}^{(\ell)}, a_{r+r'_\alpha,i}^{(\ell)}) \leftarrow (\lfloor P_{\ell'}^{-1} \cdot \bar{b} \rceil, \lfloor P_{\ell'}^{-1} \cdot \bar{a} \rceil) \in R_{Q_{\ell+1}}^2$
    $b_{r+r'_\alpha,i}^{(\ell)} \leftarrow b_{r+r'_\alpha,i}^{(\ell)} + b_{r,i}^{(\ell)}$
    $(b_{r+r'_\alpha,i}^{(\ell)}, a_{r+r'_\alpha,i}^{(\ell)}) \leftarrow (b_{r+r'_\alpha,i}^{(\ell)}(X^{5^{r'_\alpha}}), a_{r+r'_\alpha,i}^{(\ell)}(X^{5^{r'_\alpha}}))$
return $\{b_{r+r'_\alpha,i}^{(\ell)}, a_{r+r'_\alpha,i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1}$ for $\alpha = 0, \cdots, d-1$

algorithm is described in Algorithm 9. We use breath-first search when we search each node in the output arborescence. This search method is desirable for the hoisted generation of rotation keys.

5 Simulation Results with ResNet Models

In this section, we numerically verify the validity of the proposed hierarchical rotation key generation method with an appropriate computing environment for the client-server model with the ResNet standard neural network and the CKKS scheme. In the cloud computing model, the server usually has high-performance computing resources, and the client has only a general-purpose personal computer. To simulate this environment, we use a PC with Intel(R) Core(TM) i9-13900K CPU as a client and a high-performance server with NVIDIA GeForce RTX 4090 GPU accelerator as a server.

As a representative example of complex computation models, we assume that the service requested by the client requires the ResNet-20 model for the CIFAR-10 dataset or the ResNet-18 model for the ImageNet dataset. We use the parameters in Lee et al. [26], except that we use the generalized RNS decomposition method [19] in our simulation and the bit lengths of some RNS moduli are reduced with the maintained classification accuracy of the network.

Algorithm 8: HoistedPubToRot

Input: A public key $(b, a) \in R^2_{Q_{k-1}}$, d (r_α, ℓ') rotation keys, where ℓ' is higher than ℓ, $\mathsf{gk}^{(\ell')}_{r_\alpha} = \{(b^{(\ell')}_{r_\alpha,i}, a^{(\ell')}_{r_\alpha,i})\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1} \in (R^2_{Q_{\ell'}P_{\ell'}})^{\mathtt{hdnum}_{\ell'}}$ for $\alpha = 0, \cdots, d-1$, and the key level ℓ

Output: d (r_α, ℓ) rotation keys, $\mathsf{gk}^{(\ell)}_{r_\alpha} = \{b^{(\ell)}_{r_\alpha,i}, a^{(\ell)}_{r_\alpha,i}\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1} \in (R^2_{Q_\ell P_\ell})^{\mathtt{hdnum}_\ell}$ for $\alpha = 0, \cdots, d-1$

Decompose a into a vector $(a_0, \cdots, a_{\mu-1}) \in R^\mu_{P_{\ell'}Q_{\ell+1}}$, where $a_j = [a]_{Q_{\ell',j}} + Q_{\ell',j} \cdot \tilde{e}_j$ for small $\tilde{e}_j$'s for $0 \le j \le \mu - 2$ and $a_{\mu-1} = [a]_{\bar{Q}_{\ell',\mu-1}} + \bar{Q}_{\ell',\mu-1} \cdot \tilde{e}_{\mu-1}$ for small $\tilde{e}_{\mu-1}$.

for $i = 0$ **to** $\mathtt{hdnum}_\ell - 1$ **do**
- $(\bar{b}, \bar{a}) \leftarrow (0, 0) \in R^2_{P_\ell Q_\ell}$
- **for** $\alpha = 0$ **to** $d - 1$ **do**
  - **for** $j \leftarrow 0$ **to** $\mu - 1$ **do**
    - **if** $j = \mu - 1$ **then**
      - $(\bar{b}, \bar{a}) \leftarrow (\bar{b}, \bar{a}) + (a_{\mu-1} + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}^{-1}_{\ell,i}]_{Q_{\ell,i}}]_{\bar{Q}_{\ell',\mu-1}}) \cdot ([b^{(\ell')}_{r_\alpha,j}(X^{5^{-r_\alpha}})]_{P_{\ell'}Q_{\ell+1}}, [a^{(\ell')}_{r_\alpha,j}(X^{5^{-r_\alpha}})]_{P_{\ell'}Q_{\ell+1}})$
    - **else**
      - $(\bar{b}, \bar{a}) \leftarrow (\bar{b}, \bar{a}) + (a_j + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}^{-1}_{\ell,i}]_{Q_{\ell,i}}]_{Q_{\ell',j}}) \cdot ([b^{(\ell')}_{r_\alpha,j}(X^{5^{-r_\alpha}})]_{P_{\ell'}Q_{\ell+1}}, [a^{(\ell')}_{r_\alpha,j}(X^{5^{-r_\alpha}})]_{P_{\ell'}Q_{\ell+1}})$
  - $(b^{(\ell)}_{r_\alpha,i}, a^{(\ell)}_{r_\alpha,i}) \leftarrow (\lfloor P^{-1}_{\ell'} \cdot \bar{b} \rceil, \lfloor P^{-1}_{\ell'} \cdot \bar{a} \rceil) \in R^2_{Q_{\ell+1}}$
  - $b^{(\ell)}_{r_\alpha,i} \leftarrow b^{(\ell)}_{r_\alpha,i} + [b]_{Q_{\ell+1}}$
  - $(b^{(\ell)}_{r_\alpha,i}, a^{(\ell)}_{r_\alpha,i}) \leftarrow (b^{(\ell)}_{r_\alpha,i}(X^{5^{r_\alpha}}), a^{(\ell)}_{r_\alpha,i}(X^{5^{r_\alpha}}))$

return $\{b^{(\ell)}_{r_\alpha,i}, a^{(\ell)}_{r_\alpha,i}\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1}$ for $\alpha = 0, \cdots, d-1$

The sparse-secret encapsulation method [4] is assumed to be used for the bootstrapping with the dense secret key with more reduced running time and higher precision. The whole rotation steps for each model are computed with the computation method in [26] [8].

The CKKS scheme is used for the simulation, and the parameters of the CKKS scheme we use for the simulation are shown in Table 1. The `lattigo` library [1] is used for the simulation, and the `CUDA` library by NVIDIA is used for GPU acceleration of the rotation key generation. The rotation key generation with the GPU processor is implemented based on [21]. In the server, algorithms for the preparation of rotation key generation are executed on the CPU processor and all actual rotation key generation is computed by the GPU processor. The running time for rotation key generation by the client, the communication amount between the client and the server, the required storage for rotation keys in the server, and the running time for rotation key generation by the server are measured and presented in this section.

[8] The specific rotation steps is shown in Appendix D.

Algorithm 9: RotKeyGen

Input: A cyclic shift set $\mathcal{T}_\ell$ for the key level ℓ, a cyclic shift generator set $\mathcal{U}_\ell$ for the key level higher than ℓ, a set of rotation keys $\mathcal{G}_{\mathcal{U}_\ell}$ for cyclic shifts in $\mathcal{U}_\ell$ in the key level higher than ℓ, and a public key $(b, a) \in R_{Q_{k-1}}$

Output: A set of rotation keys $\mathcal{G}_{\mathcal{T}_\ell}$ for a cyclic shift set $\mathcal{T}_\ell$

$V \leftarrow \mathcal{T} \cup \{0\}$
$E \leftarrow \{(v, w) | v, w \in V\}$
$w(v, w) \leftarrow$ the minimum number of elements in $\mathcal{S}$ summed to $w - v$ allowing repetition
$G' = (V, E') \leftarrow$ Edmonds' algorithm with $G = (V, E)$; `// It can be replaced with Prim's algorithm when` $\mathcal{U}_\ell$ `is symmetric around zero.`
$Q[] \leftarrow$ empty queue for nodes
$\mathcal{G}_{\mathcal{T}_\ell} \leftarrow \varnothing$
while Q is not empty **do**
    $v \leftarrow$ dequeue from Q
    $W \leftarrow$ the set of nodes adjacent to v.
    if $v = 0$ **then**
        Generate the set of rotation keys $\mathcal{G}_W = \{\mathsf{gk}_w^{(\ell)} | w \in W\}$ from (b, a) using PubToRot or HoistedPubToRot
    else
        Generate the set of rotation keys $\mathcal{G}_W = \{\mathsf{gk}_w^{(\ell)} | w \in W\}$ from $\mathsf{gk}_v^{(\ell)}$ using RotToRot or HoistedRotToRot
    $\mathcal{G}_{\mathcal{T}_\ell} \leftarrow \mathcal{G}_{\mathcal{T}_\ell} \cup \mathcal{G}_W$
    Enqueue elements in W to Q.
return $\mathcal{G}_{\mathcal{T}_\ell}$

5.1 Parameter and Simulation Setting

We now compare the various performances between the use of the conventional system and a hierarchical rotation key system. To ensure a fair comparison, we have selected parameters based on the following criteria. When determining each special modulus, we first decide the value of `dnum` to ensure optimal rotation time. Then, we set the special modulus that allows key-switching with that specific `dnum`. In the case of a two-level hierarchical rotation key system, we assume that the `dnum` or special modulus of the higher level is not considered when determining the `dnum` or special modulus for each level. In other words, the values of `dnum` and special modulus for ciphertext key-switching are set solely to optimize the key-switching process. For a three-level scheme, we relax these criteria to carefully examine trade-offs, which will be explained in detail when discussing the results for three levels.

The bit length of the special modulus is often approximately the size of $\log Q/$`dnum` when `dnum` is given, where $\log Q$ is the ciphertext modulus bit length. Therefore, we set the special modulus based on this criterion. If the sum of the ciphertext modulus and special modulus exceeds the maximum modulus bit length, we consider it an unfeasible `dnum` for the current parameters and

Table 1: Encryption parameters in the CKKS scheme for ResNet models

Parameters	ResNet-20 for CIFAR-10	ResNet-18 for ImageNet
Polynomial modulus degree	2^{16}	2^{17}
Secret key Hamming weight	2^{15}	2^{16}
Gaussian error stand. dev.	3.2	3.2
Minimum security level	128-bit	128-bit
Maximum modulus bit-length	1714	3428

only consider larger `dnum` values for this simulation. Then, the special modulus for each key level is set using the same criteria regarding the product of the ciphertext modulus and all the special modulus for the lower key level as the ciphertext modulus for the corresponding key level.

In the case of the ResNet-20 handling the CIFAR-10 dataset, the ciphertext needs to hold a maximum of 2^{14} data points, and to perform all operations of a single layer with one bootstrapping, it is desirable to use a polynomial modulus degree of $N = 2^{16}$. The maximum modulus bit length for 128-bit security is 1714 bits assuming the secret key distribution is U_3 with the formula in [32]. Thus, we ensure that the sum of ciphertext modulus bit length and all special modulus bit lengths does not exceed 1714 bits. For ResNet-20 handling the CIFAR-10 dataset, the total ciphertext modulus bit length required is 1321 bits. For all simulations for ResNet-20, we fix the bit length of the maximum modulus and the ciphertext modulus as 1714 and 1321, respectively, independent of the value of `dnum` and the total number of the key level. Table 2 shows the parameter sets to be compared. In the table, `dnum` shows the decomposition number for the conventional system and the pair of decomposition numbers $(\texttt{dnum}_0, \texttt{dnum}_1)$ for a two-level hierarchical system.

There might be questions about whether it is fair to compare the two-level system parameters with the corresponding conventional system parameters using $\log P_0 + \log P_1$ as one special modulus. For instance, in the case of B.i for two-level system parameters $(\log Q, \log P_0, \log P_1) = (1321, 333, 60)$, one can argue that the conventional scheme with $\log P_0 + \log P_1 = 393$ as a special modulus needs to be compared. However, we cannot reduce $\texttt{dnum}_0$ from 4 to 3 by using $\log P_0 + \log P_1$ as a special modulus for a one-level scheme, because a special modulus of approximately 440 bits would be required for $\texttt{dnum}_0 = 3$, which prevents achieving the 128-bit security. In fact, A.i parameters are more efficient than the claimed parameters for a one-level system because the computation amount with special modulus is reduced. In other words, the A.i parameters are the optimal one-level system parameters in terms of the key-switching operation. Since the A.i and B.i parameters are the optimal parameters in terms of the key-switching operation for one-level system and two-level system, respectively, the comparison between these parameters is valid.

In the case of the ResNet-18 handling the ImageNet dataset, the large image size makes it difficult to accommodate intermediate values in a single ciphertext.

Therefore, although it is possible to use a degree of 2^{16}, it is preferable to set the degree to 2^{17} to double the number of slots that can be accommodated in one ciphertext for optimizing bootstrapping time. The maximum modulus bit length for 128-bit security is 3428 bits assuming the secret key distribution is U_3 with the formula in [32]. For the ResNet-18 handling the ImageNet dataset, the total ciphertext modulus bit length required is 1639 bits. For all simulations for the ResNet-18, we fix the bit length of the maximum modulus and the ciphertext modulus as 3428 and 1639, respectively, independent of the value of `dnum` and the total number of the key level. Table 3 shows the parameter sets to be compared.

In the conventional scheme, the client generates all the required rotation keys and transmits them to the server. In the two-level hierarchical rotation key scheme, the client generates the 16-ary level-1 rotation key sets with both signs and transmits them to the server, where the set of the cyclic shifts is $\{\pm 1, \pm 16, \pm 256, \cdots, \pm 2^{12}\}$. Then, the server generates the required rotation keys for the ResNet models from this 16-ary level-1 rotation key set. In the three-level hierarchical rotation key scheme, the client generates only two level-2 rotation key set, where the set of the cyclic shifts is $\{1, 256\}$. In this system, we assume that there is an inactive phase between key transmissions and an active phase when the service is provided to the client. The server can generate 4-ary level-1 rotation keys using Edmonds' algorithm for faster rotation key generation in the inactive phase before the services so that level-1 rotation keys constitute a 4-ary rotation key set, where the set of the cyclic shifts is $\{\pm 1, \pm 4, \pm 16, \cdots, \pm 2^{12}, 2^{14}\}$. Then, the server generates the required level-0 rotation keys for the ResNet models from this 4-ary level-1 rotation key set just after the services are requested, which is the active phase.

Table 2: Parameter sets with the ResNet-20 for the CIFAR-10 dataset

Rotation Key Generation		`dnum`	Modulus bit length
Conventional (One-level)	A.i	4	$(\log Q_0, \log P_0) = (1321, 333)$
	A.ii	5	$(\log Q_0, \log P_0) = (1321, 273)$
	A.iii	6	$(\log Q_0, \log P_0) = (1321, 221)$
Two-level	B.i	(4, 28)	$(\log Q_0, \log P_0, \log P_1) = (1321, 333, 60)$
	B.ii	(5, 14)	$(\log Q_0, \log P_0, \log P_1) = (1321, 273, 120)$
	B.iii	(6, 9)	$(\log Q_0, \log P_0, \log P_1) = (1321, 221, 172)$

5.2 Numerical Results

Table 4 shows the number of core operations for each rotation key set generation for ResNet models using a 16-ary rotation key set for the ResNet-20 and a 4-ary and a 16-ary rotation key sets for the ResNet-18, and it shows the effectiveness

Table 3: Parameter sets with ResNet-18 for ImageNet dataset

Rotation Key Generation		`dnum`	Modulus bit length
Conventional (One-level)	C.i	1	$(\log Q_0, \log P_0) = (1639, 1639)$
	C.ii	2	$(\log Q_0, \log P_0) = (1639, 820)$
	C.iii	3	$(\log Q_0, \log P_0) = (1639, 547)$
Two-level	D.i	(1, 22)	$(\log Q_0, \log P_0, \log P_1) = (1639, 1639, 150)$
	D.ii	(2, 3)	$(\log Q_0, \log P_0, \log P_1) = (1639, 820, 820)$
	D.iii	(3, 2)	$(\log Q_0, \log P_0, \log P_1) = (1639, 547, 1093)$
Three-level	E.iii	(3, 3, 6)	$(\log Q_0, \log P_0, \log P_1, \log P_2)$ $= (1639, 547, 729, 485)$

Table 4: Number of core operations optimized by hoisted rotation key generation and Prim's algorithm

		ResNet-20 for CIFAR-10	ResNet-18 for ImageNet	
		16-ary	4-ary	16-ary
No. of rotation Keys		265	617	
RotToRot	Total	372	649	721
	Decompose	292	347	529
PubToRot	Total	7	14	8
	Decompose	1	1	1

of the hoisted rotation key generation and Prim's algorithm. Note that the total numbers of RotToRot and PubToRot operations are close to the number of rotation keys. Roughly speaking, 1.43 numbers of key-switching operations for a rotation key are needed on average if we use a 16-ary level-1 rotation key set for the ResNet-20, and 1.07 and 1.18 numbers of key-switching operations for a rotation key are needed on average if we use 4-ary and 16-ary level-1 rotation key set for ResNet-18, respectively. It means that most rotation keys can be generated by only one RotToRot or PubToRot operation from other rotation keys, resulting from Prim's algorithm.

Note that the number of the decompose processes is effectively reduced compared to the total numbers of RotToRot and PubToRot operations by the hoisted rotation key generation. The decompose processes are the most time-consuming process in the key-switching operation. If we do not use the hoisted rotation key generation method, the number of the decompose processes is the same as the total number of RotToRot and PubToRot operations. For example, in the A.i parameter, it takes 19.0s to generate all level-0 rotation keys using 16-ary level-1 rotation keys if we do not use the hoisted method. If we use the hoisted method, it takes 16.6s to generate all level-0 rotation keys with the same level-1 rotation keys, which is reduced by 12.8%.

Table 5: Simulation results with various parameters with the ResNet-20 for the CIFAR-10 dataset

		CKR(ms)	KGKR(ms)	KGR(C)(s)	CA(GB)	KGR(S)(s)
Conventional	A.i	3.10	N/A	88.7	38.3	N/A
	A.ii	3.38	N/A	109.5	46.6	N/A
	A.iii	3.67	N/A	127.4	54.3	N/A
Two-level	B.i	3.10	12.58	18.9	8.31	16.6
	B.ii	3.38	7.21	9.68	4.16	11.7
	B.iii	3.67	5.18	6.30	2.67	10.2

Table 6: Simulation results with various parameters with the ResNet-18 for the ImageNet dataset

		CKR(ms)	KGKR(ms)	KGKR2(ms)	KGR(C)(s)	CA(GB)	OKGR(S)(s)	KGR(S)(s)
Conventional	C.i	6.50	N/A	N/A	96.1	74.7	N/A	N/A
	C.ii	6.05	N/A	N/A	145.1	115.7	N/A	N/A
	C.iii	6.59	N/A	N/A	207.9	159.1	N/A	N/A
Two-level	D.i	6.50	40.78	N/A	29.1	22.3	N/A	25.0
	D.ii	6.05	9.89	N/A	3.74	2.91	N/A	12.5
	D.iii	6.59	8.30	N/A	2.52	1.97	N/A	16.1
Three-level	E.iii	6.59	9.02	15.62	1.93	1.54	26.9	13.9

Table 5 shows the various performances with the ResNet-20 for the CIFAR-10 dataset when using the parameters in Table 2. Similarly, Table 6 shows the performances with the ResNet-18 for the ImageNet dataset when using the parameters in Table 3. Each column has the following meanings.

- CKR: Ciphertext key-switching runtime
- KGKR: Key generation key-switching runtime (level-1 $\rightarrow$ level-0)
- KGKR2: Key generation key-switching runtime (level-2 $\rightarrow$ level-1)
- KGR(C): Key generation runtime (client)
- CA: Communication amount
- OKGR(S): Key generation runtime (server, offline)
- KGR(S): Key generation runtime (server)

We present Tables 7 and 8 for both cases when the rotation keys can be predetermined before the services are provided (Deter.) and when the rotation keys are determined just at the time of the service (Non-Deter.). The term "offline" means when the service is not yet provided, and the term "online" means when the service is being provided. If the types of rotation keys are determined in advance, it is desirable to generate and transmit keys when both the conventional and proposed hierarchical systems are offline. If it is not predetermined, a conventional system cannot do anything offline, but it should generate and transmit a large key online, which is a huge burden to the client. On the other hand, in the proposed hierarchical system, if only a few highest-level rotation keys are generated and transmitted offline, communication costs do not occur online, and

Table 7: Simulation results with different situations with the ResNet-20 for the CIFAR-10 dataset. (KU: key-switching unchanged / EG: efficient generation of rotation keys)

		Runtime of client		Comm. amount		Runtime of server	
		Offline	Online	Offline	Online	Offline	Online
Deter.	Conventional	88.7s	-	38.3GB	-	-	-
	Two-level (KU)	18.9s	-	8.31GB	-	16.6s	-
	Two-level (EG)	6.30s	-	2.67GB	-	10.2s	-
Non-Deter.	Conventional	-	88.7s	-	38.3GB	-	-
	Two-level (KU)	18.9s	-	8.31GB	-	-	16.6s
	Two-level (EG)	6.30s	-	2.67GB	-	-	10.2s

Table 8: Simulation results with different situations with the ResNet-18 for the ImageNet dataset (KU: key-switching unchanged / EG: efficient generation of rotation keys)

		Runtime of client		Comm. amount		Runtime of server	
		Offline	Online	Offline	Online	Offline	Online
Deter.	Conventional	145.1s	-	115.7GB	-	-	-
	Two-level (KU)	3.74s	-	2.91GB	-	12.5s	-
	Two-level (EG)	2.52s	-	1.97GB	-	16.1s	-
	Three-level	1.93s	-	1.54GB	-	40.8s	-
Non-Deter.	Conventional	-	145.1s	-	115.7GB	-	-
	Two-level (KU)	3.74s	-	2.91GB	-	-	12.5s
	Two-level (EG)	2.52s	-	1.97GB	-	-	16.1s
	Three-level	1.93s	-	1.54GB	-	26.9s	13.9s

only a small runtime in the server is required. In other words, it shows that the computational and communication burden of the client is significantly reduced, and a large part of the computations goes to the high-performance server, which balances the computation tasks and the communication amount according to the environment. In Table 7, the parameters of the conventional system, the two-level system with key-switching unchanged, and the two-level system with more efficient rotation key generation are A.i, B.i, and B.iii, respectively. In Table 8, the parameters of the conventional system, the two-level system with key-switching unchanged, the two-level system with more efficient rotation key generation, and the three-level system are C.ii, D.ii, D.iii, and E.iii, respectively.

5.3 Discussion

Discussion of two-level system If optimizing the key-switching operation time for ciphertext is a top priority when setting parameters, firstly, it is desirable to set $\texttt{dnum}_0$ to optimize the key-switching runtime and then set $\texttt{dnum}_1$ that is possible with the surplus modulus. For example, in the case of the ResNet-20,

the optimal dnum value for key-switching operations is 4 when considering the key-switching operation time. Therefore, irrespective of whether a hierarchical rotation key is used or not, $\texttt{dnum}_0$ would be chosen as 4. In this setting, we can compare A.i and B.i parameters about the usage of the two-level hierarchical rotation key system. While the ciphertext key-switching runtime remains the same at 3.10ms, the client-side rotation key generation significantly reduces from 88.7s to 18.9s ($\times$4.69 faster), communication amount for rotation keys reduces from 38.3GB to 8.31GB ($\times$1/4.61), and server-side rotation key generation takes 16.6s. In the case of the ResNet-18 parameters, the optimal $\texttt{dnum}_0$ for key-switching operations is 2. Therefore, we can compare C.ii and D.ii. While the ciphertext key-switching runtime remains the same at 6.05ms, the client-side rotation key generation significantly reduces from 145.1s to 3.74s ($\times$38.8 faster), the communication amount for rotation keys reduces from 115.7GB to 2.91GB ($\times$1/39.8), and server-side rotation key generation takes 12.5s.

In some cases, reducing the burden on the client significantly can be more critical than the service delay. Then, we may need to make the rotation key generation more efficient at the expense of slightly increasing the key-switching operation time. For the ResNet-20 model, we can compare A.i and B.iii parameters. In other words, we can slightly increase the value of $\texttt{dnum}_0$ to set a lower $\log P_0$ and then optimize $\texttt{dnum}_1$ or $\log P_1$ for more efficient key generation. While the ciphertext key-switching runtime increases from 3.10ms to 3.67ms ($\times$1.18 slower), the client-side rotation key generation more significantly reduces from 88.7s to 6.30s ($\times$14.1 faster), the communication amount for rotation keys reduces from 38.3GB to 2.67GB ($\times$1/14.3), and server-side rotation key generation takes 10.2s. Therefore, if the actual service execution time is not highly sensitive, using A.ii two-level parameters or A.iii two-level parameters to reduce the client's burden can be advantageous. For the ResNet-18 model, we can compare C.ii and D.iii parameters. While the ciphertext key-switching runtime increases from 6.05ms to 6.59ms ($\times$1.08 slower), the client-side rotation key generation more significantly reduces from 145.1s to 2.52s ($\times$57.6 faster), the communication amount for rotation keys reduces from 115.7GB to 1.97GB ($\times$1/58.7), and server-side rotation key generation takes 16.1s. Using D.iii parameters to reduce the client's burden can be more advantageous than D.ii.

Discussion of three-level system Three-level usage can be advantageous in scenarios where the client has sent computation keys to the server but the computation model has not yet been agreed upon. In such cases, if there is a waiting time for the computation model to be decided, the runtime required to generate rotation keys for the corresponding model can be included in the online latency. Therefore, reducing the time to generate level-0 rotation keys can be important. However, the D.iii parameters have a larger server-side rotation key generation runtime than the D.ii parameters. Also, in a two-level system, to reduce the time for generating level-0 rotation keys, we may need to send more level-1 rotation keys, increasing the client's burden. For example, we may need to send a 4-ary level-1 rotation key set instead of a 16-ary level-1 rotation key set,

which would roughly double the rotation key generation time and communication amount on the client's side. However, by using a three-level rotation key, we can eliminate this trade-off.

For instance, we can compare C.ii parameters with E.iii three-level parameters. While the ciphertext key-switching runtime increases from 6.05ms to 6.59ms (x1.08 slower), the client-side rotation key generation more significantly reduces from 145.1s to 1.93s (x75.2 faster), the communication amount for rotation keys reduces from 115.7GB to 1.54GB (x75.1 less), and server-side rotation key generation takes 13.9s. In this scenario, the server needs 26.9s to prepare more level-1 rotation keys while waiting for the computation model to be decided. It allows less client-side rotation key generation and less communication with the similar server-side rotation key generation runtime to the D.ii parameters. Thus, except for the offline server-side rotation key generation, the three-level scheme shows nearly optimal online performance overall in the rotation key generation.

6 Conclusion

We proposed a hierarchical rotation key system for the CKKS and BFV schemes to significantly reduce the computational and communication costs of the client and to make the rotation key management with the reduced memory in the server. It allows the server to generate the rotation keys for the required cyclic shifts using the rotation keys in the higher key level without a secret key or any help from the clients. It can be an important future work that designs a systematic method to perform complex services with limited memory by using the proposed method more efficiently or a fast algorithm for generating a sequence of rotation keys closer to the optimal solution.

Acknowledgements This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korea Government[Ministry of Science and ICT (MSIT)]), Development of Highly Efficient Post-Quantum Cryptography (PQC) Security and Performance Verification for Constrained Devices under Grant 2021-0-00400, and in part by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (No. 2022R1I1A1A01-06828412), and in part the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2011082).

We would like to express our gratitude to the anonymous reviewers who provided insightful suggestions for effective experiments highlighting the utility of the techniques in this paper.

References

1. Lattigo v3. Online: https://github.com/tuneinsight/lattigo (Apr 2022), ePFL-LDS, Tune Insight SA

2. Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: A full rns variant of fv like somewhat homomorphic encryption schemes. In: International Conference on Selected Areas in Cryptography. pp. 423–442. Springer (2016)
3. Bossuat, J.P., Mouchet, C., Troncoso-Pastoriza, J., Hubaux, J.P.: Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In: *EUROCRYPT 2021* (2021)
4. Bossuat, J.P., Troncoso-Pastoriza, J., Hubaux, J.P.: Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In: *ACNS 2022*. pp. 521–541. Springer (2022)
5. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: *CRYPTO 2012*. pp. 868–886. Springer (2012)
6. Çetin, G.S., Chen, H., Laine, K., Lauter, K., Rindal, P., Xia, Y.: Private queries on encrypted genomic data. *BMC Medical Genomics* **10**(2), 1–14 (2017)
7. Chen, H., Chillotti, I., Song, Y.: Improved bootstrapping for approximate homomorphic encryption. In: *EUROCRYPT 2019*. pp. 34–54. Springer (2019)
8. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. In: *EUROCRYPT 2018*. pp. 360–384. Springer (2018)
9. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full RNS variant of approximate homomorphic encryption. In: *Proceedings of International Conference on Selected Areas in Cryptography (SAC)*. pp. 347–368 (2018)
10. Cheon, J.H., Hhan, M., Hong, S., Son, Y.: A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret LWE. *IEEE Access* **7**, 89497–89506 (2019)
11. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: *ASIACRYPT 2017*. pp. 409–437 (2017)
12. Cho, J., Ha, J., Kim, S., Lee, B., Lee, J., Lee, J., Moon, D., Yoon, H.: Transciphering framework for approximate homomorphic encryption. In: *ASIACRYPT 2021*. pp. 640–669. Springer (2021)
13. Edmonds, J.: Optimum branchings. *Journal of Research of the National Bureau of Standards B* **71**(4), 233–240 (1967)
14. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptol. ePrint Arch. Tech. Rep. 2012/144
15. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: *Proceedings of International Conference on Machine Learning (ICML)*. pp. 201–210. PMLR (2016)
16. Ha, J., Kim, S., Lee, B., Lee, J., Son, M.: Rubato: Noisy ciphers for approximate homomorphic encryption. In: *EUROCRYPT 2022*. pp. 581–610. Springer (2022)
17. Halevi, S., Shoup, V.: Algorithms in HElib. In: *CRYPTO 2014*. pp. 554–571. Springer (2014)
18. Halevi, S., Shoup, V.: Faster homomorphic linear transformations in helib. In: *CRYPTO 2018*. pp. 93–120. Springer (2018)
19. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: *Cryptographers' Track at the RSA Conference*. pp. 364–390. Springer (2020)
20. Jiang, X., Kim, M., Lauter, K., Song, Y.: Secure outsourced matrix computation and application to neural networks. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. pp. 1209–1222 (2018)
21. Jung, W., Kim, S., Ahn, J.H., Cheon, J.H., Lee, Y.: Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2021**(4), 114–148 (Aug 2021)

22. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A low latency framework for secure neural network inference. In: *Proceedings of the 27th USENIX Security Symposium*. pp. 1651–1669 (2018)
23. Kim, M., Lauter, K.: Private genome analysis through homomorphic encryption. In: BMC Medical informatics and decision making. vol. 15, pp. 1–12. *BioMed Central* (2015)
24. Kim, M., Song, Y., Li, B., Micciancio, D.: Semi-parallel logistic regression for gwas on encrypted data. *BMC Medical Genomics* **13**(7), 1–13 (2020)
25. Kocabas, O., Soyata, T.: Towards privacy-preserving medical cloud computing using homomorphic encryption. In: *Virtual and Mobile Healthcare: Breakthroughs in Research and Practice*, pp. 93–125. IGI Global (2020)
26. Lee, E., Lee, J.W., Lee, J., Kim, Y.S., Kim, Y., No, J.S., Choi, W.: Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In: *International Conference on Machine Learning (ICML)*. pp. 12403–12422. PMLR (2022)
27. Lee, J.W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.S., No, J.S.: Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access* **10**, 30039–30054 (2022)
28. Lee, J.W., Lee, E., Lee, Y., Kim, Y.S., No, J.S.: High-precision bootstrapping of rns-ckks homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In: *EUROCRYPT 2021*. pp. 618–647 (2021)
29. Lee, Y., Lee, J.W., Kim, Y.S., Kang, H., No, J.S.: High-precision and low-complexity approximate homomorphic encryption by error variance minimization. In: *EUROCRYPT 2022*. pp. 551–580. Springer (2022)
30. Lindell, Y.: How to simulate it–a tutorial on the simulation proof technique. Tutorials on the Foundations of Cryptography pp. 277–346 (2017)
31. Meftah, S., Tan, B.H.M., Aung, K.M.M., Yuxiao, L., Jie, L., Veeravalli, B.: Towards high performance homomorphic encryption for inference tasks on cpu: An mpi approach. *Future Generation Computer Systems* (2022)
32. Mono, J., Marcolla, C., Land, G., Güneysu, T., Aaraj, N.: Finding and evaluating parameters for bgv. In: AFRICACRYPT 2023. pp. 370–394. Springer (2023)
33. Prim, R.C.: Shortest connection networks and some generalizations. *The Bell System Technical Journal* **36**(6), 1389–1401 (1957)

A Preliminaries

A.1 Notations

Let $\mathbb{Z}, \mathbb{R}$, and $\mathbb{C}$ denote the set of integers, the set of real numbers, and the set of complex numbers, respectively. Let $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$, and let R and R_p denote the rings $\mathbb{R}[X]/(X^N+1)$ and $\mathbb{Z}_p[X]/(X^N+1)$, respectively. χ denotes an error distribution with a small variance, such as a Gaussian distribution. For integer x and positive integer p, $[x]_p$ denotes the non-negative remainder r such that $x = pq + r$, where q is an integer and $0 \leq r \leq p-1$. For integer x coprime with p, $[x^{-1}]_p$ denotes the inverse element of $[x]_p$ in $\mathbb{Z}_p$.

A.2 CKKS and BFV Schemes

Fully homomorphic encryption (FHE), which may be abbreviated as homomorphic encryption, is an encryption scheme designed to enable arbitrary arithmetic operations on encrypted data. FHE was initially defined as a bit-wise encryption scheme capable of performing all boolean operations while encrypted. The definition was mitigated to include word-wise encryption schemes capable of arbitrary arithmetic operations for encrypted integers or complex number data. The FHEs covered in this paper are CKKS and BFV schemes. These FHE schemes support arithmetic operations with the SIMD manner, allowing multiple independent data to be encrypted and operated at once in a single ciphertext with a single homomorphic operation. In the case of CKKS, the data storing structure is a one-dimensional vector, and in the case of BFV, it is a matrix in which the number of rows is two. In addition, complex numbers are encrypted in the case of CKKS, and integers are encrypted in the case of BFV. Although each scheme has several variants, we will address the schemes with the following encoding and encryption methods, where $(b, a) \in R_Q^2$ in a ring-LWE sample such that $b = -a \cdot s + e$ for the secret key $s \leftarrow \chi$ and a noise $e \leftarrow \chi$.

- CKKS scheme: The packing structure is a vector of length $N/2$, $(v_i) \in \mathbb{R}^{N/2}$. Let ζ be a $2N$-th root of unity in $\mathbb{C}$. We then obtain $m(X) \in \mathbb{R}[X]/(X^N+1)$ such that $m(\zeta^{\alpha_i}) = v_i$ for $\alpha_i = 5^j \mod 2N$ and encrypt it as $u \cdot (b, a) + (\lfloor \Delta \cdot m \rceil + e_0, e_1)$, where $u, e_0, e_1 \leftarrow \chi$ and Δ is scaling factor that determines the precision of m.
- BFV scheme: The packing structure is a $2 \times N/2$-matrix $(v_{ij}) \in \mathbb{Z}_t^{2\times N/2}$. Let ω be a $2N$-th root of unity in $\mathbb{Z}_t$. We then obtain $m(X) \in R_t$ such that $m(\omega^{\alpha_{ij}}) = v_{ij}$ for $\alpha_{ij} = (-1)^i \cdot 5^j \mod 2N$ and encrypt as $u \cdot (b, a) + (Q/t \cdot m + e_0, e_1)$, where $u, e_0, e_1 \leftarrow \chi$.

We assume that the residue number system (RNS) variants of CKKS and BFV schemes [2, 9] are used. In this variant, the ciphertext modulus is chosen as the product of large primes, and the ciphertext is represented as a vector of remainders for the primes rather than one large remainder for the ciphertext modulus. By the Chinese remainder theorem (CRT), each vector of remainders

Algorithm 10: ModUp

Input: Two disjoint sets of primes $\mathcal{C} = \{q_0, \cdots, q_{\sigma-1}\}$, $\mathcal{B} = \{p_0, \cdots, p_{\tau-1}\}$, where $Q = \prod_i q_i$ and $P = \prod_j p_j$, and an RNS-form ring element $(a_0, \cdots, a_{\sigma-1}) \in \prod_{i=0}^{\sigma-1} R_{q_i}$ for $a \in R_Q$, where $a_i = a \mod q_i$.

Output: an RNS-form ring element $(\bar{a}_0, \cdots, \bar{a}_{\sigma-1}, \tilde{a}_0, \cdots, \tilde{a}_{\tau-1}) \in \prod_{i=0}^{\sigma-1} R_{q_i} \times \prod_{j=0}^{\tau-1} R_{p_i}$ for $a' \in R_{PQ}$, where $\bar{a}_i = a' \mod q_i$, $\tilde{a}_j = a' \mod p_j$, and $a' = a + Q \cdot e$ for small e.

```
INTT operation to (a_0, ..., a_{σ-1}).
for i ← 0 to σ − 1 do
    ā_i ← a_i
    b_i ← a_i · [q̂_i^{-1}]_{q_i} ∈ R_{q_i}
for j ← 0 to τ − 1 do
    ã_j ← 0
    for i ← 0 to σ − 1 do
        ã_j ← ã_j + b_i · [q̂_i]_{p_j} ∈ R_{p_j}
NTT operation to (ā_0, ..., ā_{σ-1}, ã_0, ..., ã_{τ-1}).
return (ā_0, ..., ā_{σ-1}, ã_0, ..., ã_{τ-1}) ∈ ∏_{i=0}^{σ-1} R_{q_i} × ∏_{j=0}^{τ-1} R_{p_i}
```

for the primes has a one-to-one correspondence to the large remainder of the large modulus. The element-wise addition and multiplication between two vectors of remainders also correspond to those between two corresponding remainders of the product of the primes. The non-trivial operations in these RNS variants are the ModUp and ModDown operations. The ModUp operation raises the modulus with remaining the remainder, and the ModDown operation divides the modulus and the remainder by the product of some prime moduli and round the output. These operations include many NTT/INTT operations and CRT merge processes, one of the most time-consuming low-level operations in the CKKS and BFV. Since the decomposition process in the key-switching operation requires several ModUp processes, reducing the decomposition process is important in minimizing homomorphic operations. The specific ModUp and ModDown operations are described in Algorithms 10 and 11, where we assume that each ring element is in the NTT form.

This pair of ring elements (b, a) is the public key of each scheme. Although evaluation keys for homomorphic operations are open to the public domain, we only represent these pairs (b, a) for the encryption process as the public key in this paper. For the CKKS scheme, the level of a ciphertext is the maximum number of multiplications that can be performed on the ciphertext without bootstrapping. In the RNS-CKKS scheme, if the level of a ciphertext is ℓ, there is $\ell + 1$ number of RNS moduli for the ciphertext. Each size of RNS moduli is determined by the required precision of the multiplication in each level. Since the other homomorphic operations rather than the rotation operation of the CKKS and BFV schemes are not relevant to understanding this paper, we only

Algorithm 11: ModDown

Input: Two disjoint sets of primes $\mathcal{C} = \{q_0, \cdots, q_{\sigma-1}\}$, $\mathcal{B} = \{p_0, \cdots, p_{\tau-1}\}$, where $Q = \prod_i q_i$ and $P = \prod_j p_j$, and an RNS-form ring element $(\bar{a}_0, \cdots, \bar{a}_{\sigma-1}, \tilde{a}_0, \cdots, \tilde{a}_{\tau-1}) \in \prod_{i=0}^{\sigma-1} R_{q_i} \times \prod_{j=0}^{\tau-1} R_{p_i}$ for $a \in R_{PQ}$, where $\bar{a}_i = a \mod q_i$, $\tilde{a}_j = a \mod p_j$.

Output: an RNS-form ring element $(a'_0, \cdots, a'_{\sigma-1}) \in \prod_{i=0}^{\sigma-1} R_{q_i}$ for $a' \in R_Q$, where $a'_i = a' \mod q_i$ and $a' = \lfloor P^{-1} \cdot a \rceil + e$ for small e.

for $i \leftarrow 0$ **to** $\sigma - 1$ **do**
  $\bar{b}_i \leftarrow \bar{a}_i + [\lfloor P/2 \rfloor]_{q_i} \in R_{q_i}$
for $j \leftarrow 0$ **to** $\tau - 1$ **do**
  $\tilde{b}_j \leftarrow \tilde{a}_j + [\lfloor P/2 \rfloor]_{p_j} \in R_{p_j}$
$(\bar{b}'_0, \cdots, \bar{b}'_{\sigma-1}, \tilde{b}'_0, \cdots, \tilde{b}'_{\tau-1}) \leftarrow$ ModUp for $(\tilde{b}_0, \cdots, \tilde{b}_{\tau-1})$ from $\prod_{j=0}^{\tau-1} R_{p_i}$ to $\prod_{i=0}^{\sigma-1} R_{q_i} \times \prod_{j=0}^{\tau-1} R_{p_i}$
for $i \leftarrow 0$ **to** $\sigma - 1$ **do**
  $a'_i \leftarrow [P^{-1}]_{q_i} \cdot (\bar{b}_i - \bar{b}'_i) \in R_{q_i}$
return $(a'_0, \cdots, a'_{\sigma-1}) \in \prod_{i=0}^{\sigma-1} R_{q_i}$

deal with the rotation operation for $m(X)$ corresponding to the cyclic shift of the message vector. Detailed explanations of other operations can be found in [5, 9, 11, 14].

A.3 Key-Switching Operation and Rotation Key

We now explain the key-switching operation [19] in CKKS and BFV schemes. This operation converts a ciphertext (b, a) that can be decrypted by a secret key s to another ciphertext (b', a') that can be decrypted by another secret key s' without changing the messages. It requires an evaluation key called the key-switching key, which is constructed as follows. Suppose that we want to perform the key-switching operation switching the secret key from s to s'. The RNS moduli that we use for key-switching are Q_i for $i = 0, \cdots, \texttt{dnum} - 1$ and the special modulus is P, where `dnum` is defined to be the number of the RNS moduli decomposed for the key-switching operation. In this case, the RNS bases for these RNS moduli are $\hat{Q}_i \cdot [\hat{Q}_i^{-1}]_{Q_i}$, where $\hat{Q}_i$ means $\prod_{j \neq i} Q_j$. The special modulus P should be set to be larger than all Q_i's because of the noise reduction in the key-switching operation. We construct the key-switching key as `dnum` ciphertexts, each of which is $(b_i, a_i) \in R_{PQ}^2$, where $a_i \leftarrow R_{PQ}$ and $b_i = -a_i \cdot s' + e + P \cdot \hat{Q}_i \cdot [\hat{Q}_i^{-1}]_{Q_i} \cdot s$. In the key-switching operation, a is first decomposed into the RNS elements of a with the ModUp operation, which is described in Algorithm 12. Each RNS element is multiplied by the ciphertext having the corresponding RNS basis in the key-switching key and added with each other. Then, we divide the ciphertext and the modulus by the special modulus with the ModDown operation. The whole algorithm for key-switching operation is described in Algorithm 13. This process of temporarily raising and reducing the modulus prevents the noise

from amplifying, and the special modulus should be larger than all of the `dnum` RNS moduli used in the key-switching operation.

Algorithm 12: Decompose

Input: A ring element $a \in R_Q$ in the RNS form, where $Q = \prod_{i=0}^{\delta-1} Q_i$ and Q_i's are pairwisely coprime, and the additional modulus P coprime to Q.
Output: A vector of ring elements $(a_0, \cdots, a_{\delta-1}) \in R_{PQ}^{\delta}$, where $a_i = [a]_{Q_i} + Q_i \cdot \tilde{e}_i$ for small $\tilde{e}_i$'s and a_i's are in the RNS form.

for $i \leftarrow 0$ **to** $\delta - 1$ **do**
  $a_i \leftarrow \mathsf{ModUp}$ for $[a]_{Q_i} \in R_{Q_i}$ from R_{Q_i} to R_{PQ}.
return $(a_0, \cdots, a_{\delta-1}) \in R_{PQ}^{\delta}$

A trade-off for various performances occurs depending on the value of `dnum`. As the value of `dnum` increases, the computation amount in the key-switching operation increases due to the increase in the number of NTT/INTT operations and the amount of inner-product computation. Also, the size of the key-switching keys increases because the number of ciphertexts in the key-switching key is `dnum`. On the other hand, if the value of `dnum` is large, each RNS modulus used in the key-switching operation is small, making the special modulus small. Since the upper bound of the size of the total modulus is fixed with the specified security level, the available modulus for homomorphic computations, except the special modulus, can be large. This can accommodate a more deep homomorphic circuit without the bootstrapping operation or reduce the number of the bootstrapping operations when we perform a deep homomorphic circuit with the bootstrapping operations. The value of `dnum` is selected in consideration of these trade-offs.

If we want to perform the rotation operation for cyclic shift r, the key-switching key for this operation can be constructed as above for $s' = s(X^{5^r})$. We will call this key-switching key a rotation key for cyclic shift r of the corresponding message vector because this key is used for performing Galois automorphism $m(X) \mapsto m(X^{5^k})$ to encrypted message polynomial, which is equivalent to the rotation operations. The specific algorithm for the key-switching operation is shown in Algorithm 13.

Note that in this algorithm, we deal with a general case when the modulus $\bar{Q}$ of a ciphertext is a divisor of the maximum evaluation modulus Q. We can simply replace Q with $\bar{Q}$ in the key-switching operation with the same decomposed RNS moduli except the last RNS modulus. The non-trivial point is that

$$\{([b^{(i)}]_{P\bar{Q}}, [a^{(i)}]_{P\bar{Q}})\}_{i=0,\cdots,\mu-1} \in (R_{P\bar{Q}})^{\mu}$$

is a valid rotation key for the evaluation modulus $\bar{Q}$ and the special modulus P. For ease of understanding, we add the proof for this fact in the following theorem.

Theorem 3. *Assume that*

$$\{(b^{(i)}, a^{(i)})\}_{i=0,\cdots,\mathtt{dnum}-1} \in (R_{PQ})^{\mathtt{dnum}}$$

is a valid shift-r rotation key for the evaluation modulus Q and the special modulus P. Let $\bar{Q} = (\prod_{i=0}^{\mu-2} Q_i)\cdot\bar{Q}_{\mu-1}$, where $\bar{Q}_{\mu-1}$ is a divisor of $Q_{\mu-1}$ and $\mu \leq \mathtt{dnum}$. Then, the shift-r rotation key

$$\{([b^{(i)}]_{P\bar{Q}}, [a^{(i)}]_{P\bar{Q}})\}_{i=0,\cdots,\mu-1} \in (R_{P\bar{Q}})^{\mu}$$

is valid for the evaluation modulus $\bar{Q}$ and the special modulus P.

Proof. Since the rotation key

$$\{(b^{(i)}, a^{(i)})\}_{i=0,\cdots,\mathtt{dnum}-1} \in (R_{PQ})^{\mathtt{dnum}}$$

is valid, we have

$$b^{(i)} + a^{(i)} \cdot s = P \cdot \hat{Q}_i \cdot [\hat{Q}_i^{-1}]_{Q_i} \cdot s(X^{5^r}) + e_i \in R_{PQ}$$

for all i and small error e_i's. If we perform the modular reduction to $b^{(i)} + a^{(i)} \cdot s$ by each Q_j for $0 \leq j \leq \mu - 1$, we have

$$[b^{(i)} + a^{(i)} \cdot s]_{Q_j} = \begin{cases} [P]_{Q_i} \cdot s(X^{5^r}) + e_i & \text{if } i = j \\ e_i & \text{if } i \neq j \end{cases} \tag{1}$$

If we perform the modular reduction to $b^{(i)} + a^{(i)} \cdot s$ by each P, we have $[b^{(i)} + a^{(i)} \cdot s]_P = e_i$. Since $\bar{Q}_{\mu-1}$ is a divisor of $Q_{\mu-1}$, we can replace $Q_{\mu-1}$ in (1) with $\bar{Q}_{\mu-1}$ for all i and j.

On the other hand, we consider the following ring element

$$P \cdot \hat{\bar{Q}}_i \cdot [\hat{\bar{Q}}_i^{-1}]_{\bar{Q}_i} \cdot s(X^{5^r}) + e_i \in R_{P\bar{Q}},$$

where $\bar{Q}_i = Q_i$ for $0 \leq i \leq \mu - 2$ and $\hat{\bar{Q}}_i = \prod_{j=0, j\neq i}^{\mu-1} \bar{Q}_i$. Note that we have

$$[P \cdot \hat{\bar{Q}}_i \cdot [\hat{\bar{Q}}_i^{-1}]_{\bar{Q}_i} \cdot s(X^{5^r}) + e_i]_{\bar{Q}_j} = \begin{cases} [P]_{\bar{Q}_i} \cdot s(X^{5^r}) + e_i & \text{if } i = j \\ e_i & \text{if } i \neq j \end{cases}$$

If we perform the modular reduction to $P \cdot \hat{\bar{Q}}_i \cdot [\hat{\bar{Q}}_i^{-1}]_{\bar{Q}_i} \cdot s(X^{5^r}) + e_i$ by each P, we have $[P \cdot \hat{\bar{Q}}_i \cdot [\hat{\bar{Q}}_i^{-1}]_{\bar{Q}_i} \cdot s(X^{5^r}) + e_i]_P = e_i$.

$b^{(i)} + a^{(i)} \cdot s$ and $P \cdot \hat{\bar{Q}}_i \cdot [\hat{\bar{Q}}_i^{-1}]_{\bar{Q}_i} \cdot s(X^{5^r}) + e_i$ has the same remainders for all $\bar{Q}_i$'s and P, the two value is equal to each other in modulo $P\bar{Q}$ by the Chinese remainder theorem. Thus, we have

$$\begin{aligned} [b^{(i)}]_{P\bar{Q}} + [a^{(i)}]_{P\bar{Q}} \cdot s &= [b^{(i)} + a^{(i)} \cdot s]_{P\bar{Q}} \\ &= P \cdot \hat{\bar{Q}}_i \cdot [\hat{\bar{Q}}_i^{-1}]_{\bar{Q}_i} \cdot s(X^{5^r}) + e_i \end{aligned}$$

for all i's. Thus, the shift-r rotation key

$$\{([b^{(i)}]_{P\bar{Q}}, [a^{(i)}]_{P\bar{Q}})\}_{i=0,\cdots,\mu-1} \in (R_{P\bar{Q}})^{\mu}$$

is valid for the evaluation modulus $\bar{Q}$ and the special modulus P. □

Algorithm 13: Key-Switching Operation [19]

Input: A key-switching key from s to s',
$\mathsf{swk} = \{(b^{(i)}, a^{(i)})\}_{i=0,\cdots,\mathtt{dnum}-1} \in (R^2_{PQ})^{\mathtt{dnum}}$ for $Q = \prod_{i=0}^{\mathtt{dnum}-1} Q_i$, and a ciphertext $(b, a) \in R^2_{\bar{Q}}$ encrypted with secret key $s \in R$ for $\bar{Q} = (\prod_{i=0}^{\mu-2} Q_i) \cdot \bar{Q}_{\mu-1}$, where $\bar{Q}_{\mu-1}$ is a divisor of $Q_{\mu-1}$ and $\mu \leq \mathtt{dnum}$.
Output: A ciphertext $(b', a') \in R^2_{\bar{Q}}$ encrypted with secret key $s' \in R$

Decompose a into a vector $(a_0, \cdots, a_{\mu-1}) \in R^{\mu}_{P\bar{Q}}$, where $a_i = [a]_{Q_i} + Q_i \cdot \tilde{e}_i$ for small $\tilde{e}_i$'s for $0 \leq i \leq \mu - 2$ and $a_{\mu-1} = [a]_{\bar{Q}_{\mu-1}} + \bar{Q}_{\mu-1} \cdot \tilde{e}_{\mu-1}$ for small $\tilde{e}_{\mu-1}$.
$(\bar{b}, \bar{a}) \leftarrow (0, 0) \in R^2_{P\bar{Q}}$
for $i \leftarrow 0$ **to** $\mu - 1$ **do**
  $(\bar{b}, \bar{a}) \leftarrow (\bar{b}, \bar{a}) + a_i \cdot ([b^{(i)}]_{P\bar{Q}}, [a^{(i)}]_{P\bar{Q}})$
$(b', a') \leftarrow (\lfloor P^{-1} \cdot \bar{b} \rceil, \lfloor P^{-1} \cdot \bar{a} \rceil) \in R^2_{\bar{Q}}$
$b' \leftarrow b' + b$
return (b', a')

A.4 Graph-Theoretic Algorithms

An arborescence in a given directed graph is a directed subgraph in which a single path exists on any node from a specific root node, and a spanning arborescence is an arborescence having paths from the root node to all nodes in the graph. The minimum spanning tree problem is the problem of finding a spanning tree whose sum of edge weights is minimum. This problem is also known to be solved within polynomial time, and Edmonds' algorithm is known to solve this problem [13], shown in Algorithm 14.

A spanning tree in a given undirected graph is a subgraph in a given graph such that all edges and all nodes are connected and there is no cycle in the subgraph. The minimum spanning tree problem is the problem of finding a spanning tree whose sum of edge weights is minimum. There are many algorithms for this, but we will use Prim's algorithm [33] appropriate for the dense graph in this paper because we deal with a complete graph, shown in Algorithm 15.

B Proof of Theorem 1

Theorem 1. *The output of Algorithm 2 is a valid rotation key for the rotation operation for cyclic shift $r + r'$.*

Proof. We give the proof for $\ell' = \ell + 1$. The proof for $\ell' > \ell + 1$ is the same as the case of $\ell' = \ell + 1$ by Theorem 3. A rotation key

$$\mathsf{gk}_r^{(\ell)} = \{(b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)})\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1} \in (R_{Q_\ell P_\ell^2})^{\mathtt{hdnum}_\ell}$$

for cyclic shift r in the key level ℓ is valid if and only if

$$b_{r,i}^{(\ell)} + a_{r,i}^{(\ell)} \cdot s = P_\ell \cdot \hat{Q}_\ell \cdot [\hat{Q}_\ell^{-1}]_{Q_\ell} \cdot s(X^{5^r}) + e_{r,i}^{(\ell)}$$

Algorithm 14: Edmonds' Algorithm[13]

Input: A directed graph $G = (V, E)$ with an edge weight $w(e)$ for all $e \in E$, the root node $v_r \in V$
Output: A minimum spanning arborescence $G' = (V, E')$ from the root node v_r

Remove all edges to the root node v_r from E
$E' \leftarrow \varnothing$
for $v \in V \backslash \{v_r\}$ **do**
  $\pi(v) \leftarrow$ the node such that an edge $(\pi(v), v) \in E$ has the minimum weight among edges to v
  $E' \leftarrow E' \cup \{(\pi(v), v)\}$
if $G' = (V, E')$ has no cycle **then**
  return $G' = (V, E')$
else
  $C = (V_c, E_c) \leftarrow$ a cycle in G'
  $\bar{V} \leftarrow (V \backslash V_c) \cup \{v_c\}$ for new node v_c
  $\bar{E} \leftarrow E \backslash E_c$
  for $(v_1, v_2) \in E$ such that $v_1 \in V \backslash V_c, v_2 \in V_c$ **do**
    Generate an edge (v_1, v_c) with a weight $w(v_1, v_c) = w(v_1, v_2) - w(\pi(v_2), v_2)$
    $\bar{E} \leftarrow (\bar{E} \backslash \{(v_1, v_2)\}) \cup \{(v_1, v_c)\}$
  for $(v_1, v_2) \in E$ such that $v_1 \in V_c, v_2 \in V \backslash V_c$ **do**
    if $(v_c, v_2) \notin \bar{E}$ or $w(v_c, v_2) > w(v_1, v_2)$ **then**
      Generate (or update) an edge (v_c, v_2) with a weight $w(v_c, v_2) = w(v_1, v_2)$
      $\bar{E} \leftarrow \bar{E} \backslash \{(v_1, v_2)\} \cup \{(v_c, v_2)\}$
  $\bar{G}' = (\bar{V}, \bar{E}') \leftarrow$Edmonds' algorithm for $(\bar{V}, \bar{E})$ with v_r
  $E' \leftarrow$ all edges in E that correspond to edges in $\bar{E}'$
  $v_t \leftarrow$ the node such that $(u, v_t) \in E'$ corresponds to $(u, v_c) \in \bar{E}'$
  $E' \leftarrow (E' \cup E_c) \backslash \{(\pi(v_t), v_t)\}$
  return $G' \leftarrow (V, E')$

Algorithm 15: Prim's Algorithm[33]

Input: An undirected graph $G = (V, E)$ with an edge weight $w(e)$ for all $e \in E$
Output: A minimum spanning tree $G' = (V, E')$

Initialize $G' = (V', E')$, where $V' \leftarrow \{v\}, E' \leftarrow \varnothing$ for randomly selected $v \in V$
while $V' \neq V$ **do**
  $\bar{E} \leftarrow \{(v_1, v_2) | v_1 \in V', v_2 \in V \backslash V'\}$
  Find an edge $\bar{e} = (\bar{v}_1, \bar{v}_2) \in \bar{E}$ having the minimum edge weight among $\bar{E}$
  $V' \leftarrow V' \cup \{\bar{v}_2\}$
  $E' \leftarrow E' \cup \{\bar{e}\}$
return $G' = (V, E')$

for small errors $e_{r,i}^{(\ell)}$. If we perform

$$(\tilde{b}_{r,i}, \tilde{a}_{r,i}) \leftarrow (b_{r,i}^{(\ell)}(X^{5^{r'}}), a_{r,i}^{(\ell)}(X^{5^{r'}}))$$

as in line 2 of Algorithm 2, we have

$$\begin{aligned}\tilde{b}_{r,i} + \tilde{a}_{r,i} \cdot s(X^{5^{r'}}) &= b_{r,i}^{(\ell)}(X^{5^{r'}}) + a_{r,i}^{(\ell)}(X^{5^{r'}}) \cdot s(X^{5^{r'}}) \\ &= P_\ell \cdot \hat{Q}_\ell \cdot [\hat{Q}_\ell^{-1}]_{Q_\ell} \cdot s((X^{5^{r'}})^{5^r}) + e_{r,i}^{(\ell)}(X^{5^{r'}}) \\ &= P_\ell \cdot \hat{Q}_\ell \cdot [\hat{Q}_\ell^{-1}]_{Q_\ell} \cdot s(X^{5^{r+r'}}) + e_{r,i}^{(\ell)}(X^{5^{r'}}).\end{aligned}$$

If we perform the key-switching operation to $(\tilde{b}_{r,i}, \tilde{a}_{r,i})$ from $s(X^{5^{r'}})$ to $s(X)$ as in line 3 of Algorithm 2, the output $(b_{r+r',i}^{(\ell)}, a_{r+r',i}^{(\ell)})$ satisfies

$$\begin{aligned}b_{r+r',i}^{(\ell)} + a_{r+r',i}^{(\ell)} \cdot s &= \tilde{b}_{r,i} + \tilde{a}_{r,i} \cdot s(X^{5^{r'}}) + e'_{r,i} \\ &= P_\ell \cdot \hat{Q}_\ell \cdot [\hat{Q}_\ell^{-1}]_{Q_\ell} \cdot s(X^{5^{r+r'}}) + e_{r,i}^{(\ell)}(X^{5^{r'}}) + e'_{r,i}\end{aligned}$$

for small errors $e'_{r,i}$ generated from the key-switching operation. Since $e_{r,i}^{(\ell)}(X^{5^{r'}}) + e'_{r,i}$ is a small polynomial,

$$\mathsf{gk}_{r+r'}^{(\ell)} = \{b_{r+r',i}^{(\ell)}, a_{r+r',i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1}$$

is a valid rotation key for cyclic shift $r + r'$ in the key level ℓ.

To use the rotation key $\mathsf{gk}_{r'}^{(\ell')} \in (R^2_{Q_{\ell'}P_{\ell'}})^{\mathtt{hdnum}_{\ell'}}$ in this key-switching operation, the modulus of the ciphertext to be key-switched should be a divisor of $Q_{\ell'}$. Note that $Q_\ell = P_{\ell-1}Q_{\ell-1}$ for all ℓ. If the key level ℓ is less than ℓ', $P_\ell Q_\ell$ is a divisor of $Q_{\ell'}$. Therefore, $(\tilde{b}_{r,i}, \tilde{a}_{r,i})$ can be key-switched by $\mathsf{gk}_{r'}^{(\ell')}$.

□

C Proof of Theorem 2

Theorem 2. *The output of Algorithm 3 is a valid rotation key for the rotation operation for cyclic shift r.*

Proof. We give the proof for $\ell' = \ell + 1$. The proof for $\ell' > \ell + 1$ is the same as the case of $\ell' = \ell + 1$ by Theorem 3. A public key $(b, a) \in R^2_{Q_{k-1}}$ is valid if and only if

$$b + a \cdot s = e$$

for small $e \in R_{Q_{k-1}}$. Note that ℓ is less than $k - 1$, and $P_\ell Q_\ell$ is a divisor of Q_{k-1}. If we perform

$$(b', a') \leftarrow ([b(X^{5^r})]_{P_\ell Q_\ell}, [a(X^{5^r})]_{P_\ell Q_\ell}) \in R^2_{P_\ell Q_\ell}$$

as in line 1 of Algorithm 3, we have

$$b' + a' \cdot s(X^{5^r}) = b(X^{5^r}) + a(X^{5^r}) \cdot s(X^{5^r}) = e(X^{5^r}) \in R_{P_\ell Q_\ell}.$$

If we decompose a' into a vector $(a_0, \cdots, a_{\mathtt{hdnum}_{\ell'}-1}) \in R_{P_{\ell'}Q_{\ell'}}^{\mathtt{hdnum}_{\ell'}}$ using ModUp opeation, we have $a_j = [a']_{Q_{\ell',j}} + Q_{\ell',j} \cdot \tilde{e}_j$ for small $\tilde{e}_j$'s, rather than $[a']_{Q_{\ell',j}}$. The reason for this is the fast basis conversion technique [2], which omits the modular reduction by the product of moduli in the CRT merge process to remove the need for transforming to non-RNS representation.

On the other hand, the rotation key

$$\mathsf{gk}_r^{(\ell')} = \{b_{r,j}^{(\ell')}, a_{r,j}^{(\ell')}\}_{j=0,\cdots,\mathtt{hdnum}_{\ell'}-1} \in (R_{Q_{\ell'}P_{\ell'}}^2)^{\mathtt{hdnum}_{\ell'}}$$

for cyclic shift r in the key level ℓ' satisfies

$$b_{r,j}^{(\ell')} + a_{r,j}^{(\ell')} \cdot s = P_{\ell'} \cdot \hat{Q}_{\ell',j} \cdot [\hat{Q}_{\ell',j}^{-1}]_{Q_{\ell',j}} \cdot s(X^{5^r}) + e_{r,j}^{(\ell')}$$

for small errors $e_{r,j}^{(\ell')}$. lines 4-7 compute

$$\sum_{j=0}^{\mathtt{hdnum}_{\ell'}-1} (a_j + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}) \cdot (b_{r,j}^{(\ell')}, a_{r,j}^{(\ell')}) \in R_{Q_{\ell'}P_{\ell'}}^2,$$

which we denote as $(\bar{b}_{r,i}^{(\ell)}, \bar{a}_{r,i}^{(\ell)})$. The term $a_j + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}$ can be arranged as

$$\begin{aligned}
&a_j + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}} \\
=&[a']_{Q_{\ell',j}} + Q_{\ell',j} \cdot \tilde{e}_j + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}} \\
=&[a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}} + Q_{\ell',j} \cdot e'_{i,j} + Q_{\ell',j} \cdot \tilde{e}_j \\
=&[a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}} + Q_{\ell',j} \cdot (e'_{i,j} + \tilde{e}_j),
\end{aligned}$$

where $Q_{\ell',j} \cdot e'_{i,j}$ denotes the difference between $[a']_{Q_{\ell',j}} + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}$ and $[a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}$. The difference occurs because these operations are performed in $R_{P_{\ell'}Q_{\ell'}}$, rather than $R_{Q_{\ell',j}}$. Since $[a']_{Q_{\ell',j}}$ and $[P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}$ are positive integer polynomials less than $Q_{\ell',j}$, $[a']_{Q_{\ell',j}} + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}$ is a positive integer polynomial less than $2Q_{\ell',j}$. The polynomial $[a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}$ is a positive integer polynomial less than $Q_{\ell',j}$, and $[a']_{Q_{\ell',j}} + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}$ and $[a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}$ are the same in modulo $Q_{\ell',j}$. Thus, the difference is the multiple of $Q_{\ell',j}$ so that it has the form of $Q_{\ell',j} \cdot e'_{i,j}$, and $e'_{i,j}$ is polynomials having zero or one as its coefficients.

If we compute $\bar{b}_{r,i}^{(\ell)} + \bar{a}_{r,i}^{(\ell)} \cdot s$, we have

$$\begin{aligned}
&\bar{b}_{r,i}^{(\ell)} + \bar{a}_{r,i}^{(\ell)} \cdot s \\
&= \sum_{j=0}^{\mathtt{hdnum}_{\ell'}-1} (a_j + [P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}}) \cdot (b_{r,j}^{(\ell')} + a_{r,j}^{(\ell')} \cdot s) \\
&= \sum_{j=0}^{\mathtt{hdnum}_{\ell'}-1} ([a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}} + Q_{\ell',j} \cdot (e'_{i,j} + \tilde{e}_j)) \cdot \\
&\qquad\qquad (P_{\ell'} \cdot \hat{Q}_{\ell',j} \cdot [\hat{Q}_{\ell',j}^{-1}]_{Q_{\ell',j}} \cdot s(X^{5^r}) + e_{r,j}^{(\ell')}) \\
&= P_{\ell'} \cdot \left(\sum_{j=0}^{\mathtt{hdnum}_{\ell'}-1} [a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}} \right. \\
&\qquad\qquad \left. \cdot \hat{Q}_{\ell',j} \cdot [\hat{Q}_{\ell',j}^{-1}]_{Q_{\ell',j}} \cdot s(X^{5^r}) \right) \\
&\qquad + ([a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}]_{Q_{\ell',j}} + Q_{\ell',j} \cdot (e'_{i,j} + \tilde{e}_j)) \cdot e_{r,j}^{(\ell')} \\
&= P_{\ell'} \cdot (a' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}}) \cdot s(X^{5^r}) + E_{i,j} \\
&= P_{\ell'} \cdot (-b' + P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}} \cdot s(X^{5^r}) + e(X^{5^r})) + E_{i,j},
\end{aligned}$$

where $E_{i,j}$ denotes the remaining error term.

If we perform $\mathsf{ModDown}$ operation to $(\bar{b}_{r,i}^{(\ell)}, \bar{a}_{r,i}^{(\ell)})$ to divide the terms and the modulus by $P_{\ell'}$ and add $(b', 0)$, which we denotes $(b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)})$, we have

$$\begin{aligned}
&b_{r,i}^{(\ell)} + a_{r,i}^{(\ell)} \cdot s \\
&= P_\ell \cdot \hat{Q}_{\ell,i} \cdot [\hat{Q}_{\ell,i}^{-1}]_{Q_{\ell,i}} \cdot s(X^{5^r}) + e(X^{5^r}) + \lfloor P_{\ell'}^{-1} \cdot E_{i,j} \rceil.
\end{aligned}$$

Since we choose $P_{\ell'}$ as larger than $Q_{\ell',j}$ for all j, the term $\lfloor P_{\ell'}^{-1} \cdot E_{i,j} \rceil$ is only small error. Thus,

$$\mathsf{gk}_r^{(\ell)} = \{b_{r,i}^{(\ell)}, a_{r,i}^{(\ell)}\}_{i=0,\cdots,\mathtt{hdnum}_\ell - 1} \in (R_{Q_\ell P_\ell}^2)^{\mathtt{hdnum}_\ell}$$

is a valid rotation key for cyclic shift r in the key level ℓ.

□

D Required Rotation Keys for ResNet Models

The set of cyclic shifts required to perform the ResNet-20 for the CIFAR-10 dataset is enumerated as follows.

– $\mathcal{T}_0^{\mathsf{ResNet-20}}$ = {1, -1, 2, -2, 3, 4, -4, 5, 6, 7, 8, -8, 9, 12, 16, -16, 18, 27, 28, 32, -32, 36, 45, 48, 54, 56, 63, 64, -64, 72, 80, 84, 96, -96, 112, 128,

-128, 192, 256, 384, 512, 768, 959, 960, 990, 991, -994, 1008, 1023, 1024, -1024, -1025, 1036, -1056, 1064, -1088, 1092, -1120, 1536, 1952, 1982, 1983, 2016, 2044, 2047, 2048, -2048, 2072, 2078, -2080, 2100, -2112, -2144, 3007, 3024, 3040, 3052, 3070, 3071, 3072, -3072, 3080, -3104, 3108, -3136, -3168, 3840, 3904, 3968, 4031, 4032, 4062, 4063, 4080, 4084, 4088, 4092, 4095, 4096, -4096, 4104, -4128, -4131, -4195, 5023, 5024, 5054, 5055, 5087, 5118, 5119, 5120, -5120, -5152, -5155, -5219, 6047, 6078, 6079, 6111, 6112, 6142, 6143, 6144, -6144, -6176, -6179, -6243, 7071, 7102, 7103, 7135, 7166, 7167, 7168, -7168, -7200, -7203, -7267, 7936, 8000, 8064, 8095, 8126, 8127, 8128, 8159, 8176, 8180, 8184, 8188, 8190, 8191, 8192, -8192, -8195, 8200, -8225, -8226, -8227, -8259, -8290, -8291, 9149, 9183, 9184, 9213, 9215, 9216, -9219, -9249, -9250, -9251, -9283, -9314, -9315, 10173, 10207, 10208, 10237, 10239, 10240, -10240, -10243, -10273, -10274, -10275, -10307, -10338, -10339, 11197, 11231, 11232, 11261, 11263, 11264, -11264, -11267, -11297, -11298, -11299, -11331, -11362, -11363, 12221, 12255, 12256, 12285, 12287, 12288, -12288, -12321, -12385, 13214, 13216, 13246, 13278, 13279, 13280, 13310, 13311, 13312, -13345, -13409, 14238, 14240, 14270, 14302, 14303, 14304, 14334, 14335, 14336, -14336, -14369, -14433, 15262, 15264, 15294, 15326, 15327, 15328, 15358, 15359, 15360, -15393, -15457, 15872, 16000, 16128, 16256, 16286, 16288, 16318, 16350, 16351, 16352, 16368, 16372, 16376, -16376, 16380, 16382, 16383, 16384}

The set of cyclic shifts required to perform the ResNet-18 for the ImageNet dataset is enumerated as follows.

– $\mathcal{T}_0^{\mathsf{ResNet-18}} = \{1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6, 7, -7, 8, -8, -9, -10, -11, -12, -13, -14, -15, 16, -16, -17, -18, -19, -20, -21, -22, -23, 24, -24, -25, -26, -27, -28, -29, -30, -31, 32, -32, 39, 64, 78, 96, 117, 128, 156, 160, 192, 195, 221, 222, 224, -224, -225, -226, -227, -228, -229, -230, -231, -232, -233, 234, -234, -235, -236, -237, -238, -239, -240, -241, -242, -243, -244, -245, -246, -247, -248, -249, -250, -251, -252, -253, -254, -255, 256, -256, 273, 312, 351, 384, 390, 429, 445, 448, -448, -449, -450, -451, -452, -453, -454, -455, -456, -457, -458, -459, -460, -461, -462, -463, -464, -465, -466, -467, 468, -468, -469, -470, -471, -472, -473, -474, -475, -476, -477, -478, -479, 507, 512, -512, 546, 576, 585, 624, 663, 669, -672, -673, -674, -675, -676, -677, -678, -679, -680, -681, -682, -683, -684, -685, -686, -687, -688, -689, -690, -691, -692, -693, -694, -695, -696, -697, -698, -699, -700, -701, 702, -702, -703, 741, 768, -768, 780, 819, 858, 896, -896, 897, -897, -898, -899, -900, -901, -902, -903, -904, -905, -906, -907, -908, -909, -910, -911, -912, -913, -914, -915, -916, -917, -918, -919, -920, -921, -922, -923, -924, -925, -926, -927, 936, 960, 975, -999, 1014, 1024, -1024, 1053, 1092, -1120, -1121, -1122, -1123, -1124, -1125, -1126, -1127, -1128, -1129, -1130, 1131, -1131, -1132, -1133, -1134, -1135, -1136, -1137, -1138, -1139, -1140, -1141, -1142, -1143, -1144, -1145, -1146, -1147, -1148, -1149, -1150, -1151, 1152, 1170, 1209, 1248, 1287, 1326, 1344, -1344, -1345, -1346, -1347, -1348, -1349, -1350, -1351, -1352, -1353, -1354, -1355, -1356, -1357, -1358, -1359, -1360, -1361, -1362, -1363, -1364, 1365, -1365, -1366, -1367, -1368, -1369, -1370, -1371, -1372, -1373, -1374, -1375, 1404, 1443, 1482, 1536, -1568, -1569, -1570, -1571,

-1572, -1573, -1574, -1575, -1576, -1577, -1578, -1579, -1580, -1581, -1582, -1583, -1584, -1585, -1586, -1587, -1588, -1589, -1590, -1591, -1592, -1593, -1594, -1595, -1596, -1597, -1598, -1599, 1728, 1792, -1792, -1793, -1794, -1795, -1796, -1797, -1798, -1799, -1800, -1801, -1802, -1803, -1804, -1805, -1806, -1807, -1808, -1809, -1810, -1811, -1812, -1813, -1814, -1815, -1816, -1817, -1818, -1819, -1820, -1821, -1822, -1823, 1920, -2016, -2017, -2018, -2019, -2020, -2021, -2022, -2023, -2024, -2025, -2026, -2027, -2028, -2029, -2030, -2031, -2032, -2033, -2034, -2035, -2036, -2037, -2038, -2039, -2040, -2041, -2042, -2043, -2044, -2045, -2046, -2047, 2048, 2112, -2240, -2241, -2242, -2243, -2244, -2245, -2246, -2247, -2248, -2249, -2250, -2251, -2252, -2253, -2254, -2255, -2256, -2257, -2258, -2259, -2260, -2261, -2262, -2263, -2264, -2265, -2266, -2267, -2268, -2269, -2270, -2271, 2304, -2464, -2465, -2466, -2467, -2468, -2469, -2470, -2471, -2472, -2473, -2474, -2475, -2476, -2477, -2478, -2479, -2480, -2481, -2482, -2483, -2484, -2485, -2486, -2487, -2488, -2489, -2490, -2491, -2492, -2493, -2494, -2495, 2496, 2688, -2688, -2689, -2690, -2691, -2692, -2693, -2694, -2695, -2696, -2697, -2698, -2699, -2700, -2701, -2702, -2703, -2704, -2705, -2706, -2707, -2708, -2709, -2710, -2711, -2712, -2713, -2714, -2715, -2716, -2717, -2718, -2719, 2880, -2912, -2913, -2914, -2915, -2916, -2917, -2918, -2919, -2920, -2921, -2922, -2923, -2924, -2925, -2926, -2927, -2928, -2929, -2930, -2931, -2932, -2933, -2934, -2935, -2936, -2937, -2938, -2939, -2940, -2941, -2942, -2943, 3072, -3136, -3137, -3138, -3139, -3140, -3141, -3142, -3143, -3144, -3145, -3146, -3147, -3148, -3149, -3150, -3151, -3152, -3153, -3154, -3155, -3156, -3157, -3158, -3159, -3160, -3161, -3162, -3163, -3164, -3165, -3166, -3167, -3360, -3361, -3362, -3363, -3364, -3365, -3366, -3367, -3368, -3369, -3370, -3371, -3372, -3373, -3374, -3375, -3376, -3377, -3378, -3379, -3380, -3381, -3382, -3383, -3384, -3385, -3386, -3387, -3388, -3389, -3390, -3391, 3584, -3584, 4096, 5120, 6144, 7168, -7168, 8192, -8192, 14336, 16384, -16384, 21504, -22528, 24576, -24576, 28672, -29696, 32768}