ARK: Adaptive Rotation Key Management for Fully Homomorphic Encryption Targeting Memory Efficient Deep Learning Inference

Jia-Lin Chan, Wai-Kong Lee, *Member, IEEE,* Denis C.-K Wong
Wun-She Yap, and Bok-Min Goi, *Senior Member, IEEE*

***Abstract*—Advancements in deep learning (DL) not only revolutionized many aspects in our lives, but also introduced privacy concerns, because it processed vast amounts of information that was closely related to our daily life. Fully Homomorphic Encryption (FHE) is one of the promising solutions to this privacy issue, as it allows computations to be carried out directly on the encrypted data. However, FHE requires high computational cost, which is a huge barrier to its widespread adoption. Many prior works proposed techniques to enhance the speed performance of FHE in the past decade, but they often impose significant memory requirements, which may be up to hundreds of gigabytes. Recently, focus has shifted from purely improving speed performance to managing FHE's memory consumption as a critical challenge. Rovida and Leporati introduced a technique to minimize rotation key memory by retaining only essential keys, yet this technique is limited to cases with symmetric numerical patterns (e.g., -2 -1 0 1 2), constraining its broader utility. In this paper, a new technique, Adaptive Rotation Key (ARK), is proposed that minimizes rotation key memory consumption by exhaustively analyzing numerical patterns to produce a minimal subset of shared rotation keys. ARK also provides a dual-configuration option, enabling users to prioritize memory efficiency or computational speed. In memory-prioritized mode, ARK reduces rotation key memory consumption by 41.17% with a 12.57% increase in execution time. For speed-prioritized mode, it achieves a 24.62% rotation key memory reduction with only a 0.21% impact on execution time. This flexibility positions ARK as an effective solution for optimizing FHE across varied use cases, marking a significant advancement in optimization strategies for FHE-based privacy-preserving systems.**

***Index Terms*—Fully Homomorphic Encryption, Memory Optimization, Deep Learning, Privacy Preservation, Adaptive Rotation Key, ARK**

I. Introduction

Recent rapid advancements in deep learning (DL) have raised critical concerns regarding user privacy, especially for applications that deals with our personal information and daily lives. Such concerns are even more serious for cloud-based deployments that store and consume user data on third-party cloud servers, which is commonly regarded as "honest but curious". Fully Homomorphic Encryption (FHE) has emerged as a promising solution to such concerns, because it allows computations to be carried out on encrypted data. This prevents exposure of sensitive information during deployment of DL applications. However, large-scale DL computation using FHE is significantly slower (up to 100 million times [1]) than those without encryption, posing a major barrier to widespread adoption. This performance gap has led to active research focus on optimizing FHE, particularly for DL applications. Recent efforts have shown the potential to construct FHE-protected neural networks that can provide reasonable speed performance [2]–[4], achieving approximately 255 seconds per image for ResNet-20 inference on CIFAR-10 [4].

Despite these advancements, many FHE optimization techniques significantly increased the memory requirements, demanding up to hundreds of gigabytes [2], [4]–[6]. Recent works [7]–[10] revealed that memory is an emerging bottleneck in FHE applications. This limitation inherently restricts the application and advancement of FHE in deep learning, posing a barrier to its ability to provide secure solutions for increasingly complex challenges. In particular, [8], [9] discovered that one contributor to the high memory consumption is the substantial memory required for rotation keys during parameter setup, a challenge that intensifies with model complexity. For instance, the memory requirement is anticipated to reach terabyte scales for techniques in [3] on a standard ResNet-20. Such high memory demand results in significant computational, storage, and communication challenges. The growing complexity in DL models emphasizes the need for memory optimization strategies to efficiently support FHE-based applications in modern privacy-preserving computing.

Techniques that strike a balance between memory consumption and performance speed is becoming an emerging trend for FHE-protected deep learning. Recent studies focused on optimizing the rotation key memory usage to enhance the efficiency of FHE-based privacy-preserving systems [9], [11]. However, these approaches exhibit limitations, indicating room for further improvements in optimization. For instance, Lee et al. [9] proposed a hierarchical rotation key generation system to reduce rotation key memory. However, they do not report the impact of increasing the rotation count on execution time, which raises concerns about the overall feasibility of the approach. Subsequently, the LowMem technique introduced by [11] reduces rotation key memory by storing a subset of rotation keys for reconstruction. However, its applicability is limited to specific numerical patterns in rotation keys, particularly symmetric number numerical patterns, as detailed in Section II-A. In contrast, geometric numerical patterns (2^n) are also found in the rotation keys in the FHE domain, as

The research was supported by the Ministry of Higher Education (MoHE), Malaysia, through the Fundamental Research Grant Scheme (FRGS/1/2021/ICT07/UTAR/01/1)
Contributing authors: vicracechan@gmail.com, waikong.lee@gmail.com, deniswong@utar.edu.my, yapws@utar.edu.my, goibm@utar.edu.my;

observed in [4] and illustrated in Fig. 6 $rot2$. Additionally, ConvFHE [4] also features another set of rotation keys that do not adhere to a clear pattern and appear random. Such limitation highlights the potential for further enhancement, as a more flexible solution could improve the effectiveness of rotation key management across a broader range of scenarios. In particular, such solution should cover the symmetric numerical pattern found in the rotation key in LowMem [11], geometric numerical pattern found in the ConvFHE framework [4], as well as other unknown and irregular patterns.

In this paper, a novel technique to accommodate extensive lists of numerical pattern and generate corresponding sets to minimize the memory consumption for rotation keys, while maintaining a reasonable performance speed, is proposed. The key contributions of this work are outlined as follows:

1) **Technique Development**: A technique, ARK, is developed to minimize the number of rotation keys used in FHE and its memory consumption, harnessing the inherent regularities in numerical sequence patterns. Unlike LowMem [11] which only handles symmetric numerical patterns, the proposed ARK can handle a wide range of numerical sequence patterns and generate a minimal subset of rotation keys. This is achieved through a series of algorithms that exhaustively search the potential numerical sequence patterns, identify minimal subsets of rotation keys for each pattern, and select the optimal set of keys to cover all necessary rotations. Thus, ARK supports a broader range of numerical patterns beyond the limitations of LowMem [11]. It redefines the balance between memory efficiency and computational speed in FHE operations, achieving a smaller key storage size while ensuring comprehensive reconstruction of rotation keys. ARK demonstrates the capability to reduce the number of rotation keys by up to 48.71%, and consumes up to 57.25% lesser time in generating rotation key.
2) **Adaptive Dual-Configuration Approach**: The proposed ARK technique also features a novel dual-configuration approach, allowing users to prioritize either memory reduction or time efficiency in generating the rotation setups for FHE-based CNN. In memory-constrained scenarios, the user may opt for a configuration that substantially reduces the memory consumption. Experimental results show that we can lower the setup memory by 25.33% to 41.17%, in expense of 0.73% to 17.84% time increase. Conversely, in time-critical operations, the time-prioritized configuration minimizes time degradation to as little as 0.21%, while still delivering significant memory reductions of 18.26% to 39.41%. Notably, the technique enables near zero-cost operations when ciphertext can be pre-rotated, allowing for the use of stored intermediate results in subsequent operations without increasing the rotation count.

The proposed ARK showcases adaptive flexibility enabled by the adaptive dual configuration, marking an advancement that allows for customization to address diverse operational demands. This adaptability not only improves overall efficiency but also broadens the applicability of FHE in situations where resource constraints and performance requirements must be carefully configured to achieve a balanced performance.

II. Background

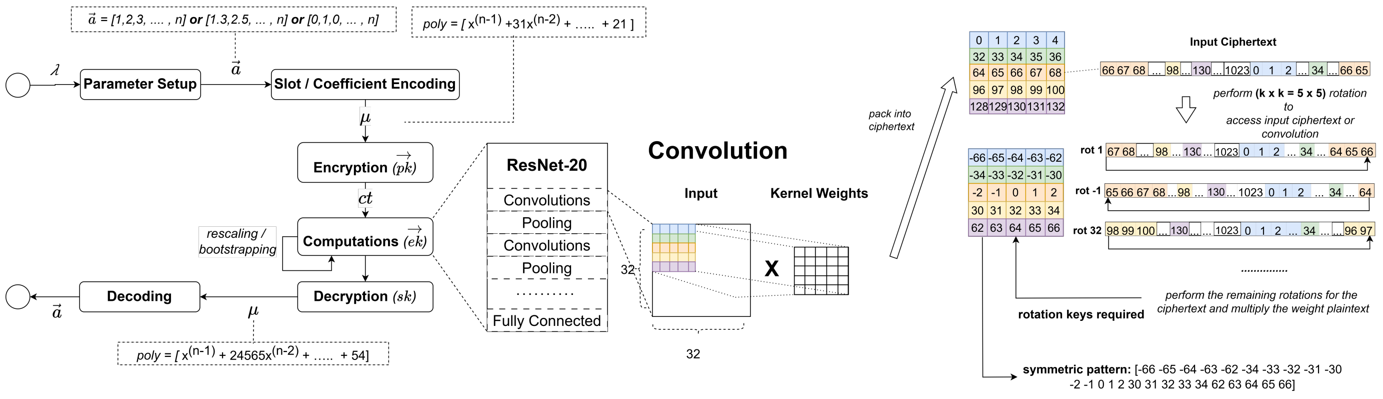
A. Overview of Fully Homomorphic Encryption

Fully Homomorphic Encryption facilitates computations on encrypted data without concerns of privacy leakage, making it a leading approach for secure data processing. Figure 1 shows the overview of computational processes for an FHE-protected application. Initially, the security level λ are determined to initialize the FHE components and keys (e.g., $\overrightarrow{pk}$, sk, $\overrightarrow{ek}$). Subsequently, the input messages ($\overrightarrow{a}$) are transformed into polynomial form, often referred to as plaintext (μ), through a process known as encoding. There are two primary types of encoding processes in FHE:

1) **Slot Encoding**: The messages are transformed into a vector through an Inverse Discrete Fourier Transform (IDFT). However, it requires rotations to aggregate the intermediate results within a single ciphertext during convolution. For instance, previous studies [11], [12] have performed a convolution by producing k^2 rotated ciphertexts, where k denotes the kernel size.
2) **Coefficient Encoding**: This method encodes messages as polynomial coefficients. Research conducted by [4] has shown that the encrypted multiplication of ciphertext leads to convolution operations, thereby reducing rotation needs during convolution (Conv2D). Nonetheless, it still requires rotations to transform the output for subsequent layers after Conv2D.

Following encoding, the plaintext can be encrypted to ciphertext (ct) for computation using the public key ($\overrightarrow{pk}$). In FHE, the operations are performed on groups of inputs, thus, rotations are necessary to align ciphertext to facilitate convolution and transform data to subsequent layers. For example, as shown in Fig. 1, in the convolution with a 32×32 feature size and a 5×5 filter. The center input (66) is used as the starting index in the ciphertext rather than the initial input index 0, thus deriving 25 rotation keys with a symmetric numerical pattern. This positioning ensures that relevant slots in the ciphertext align to yield the required results in each slot during convolution, thereby minimizing the need for more rotations to realign data. Rescaling and bootstrapping are steps to refresh the ciphertext, allowing for additional computations when the X_{noise} reaches a threshold. Once the computation is completed, ciphertext is transmitted to users, in which decryption is possible with secret key (sk). Finally, a decoding operation is executed to convert the plaintext back into a usable form for the end user.

Cheon, Kim, Kim and Song (CKKS) [13] scheme was proposed to facilitate approximate computations and support real numbers, which makes them more suitable for DL applications compared to Brakerski/ Fan-Vercauteren (BFV) [14], [15] and Brakerski-Gentry-Vaikuntanathan (BGV) [16] schemes that restricted to integer representations. Although this advancement provided greater flexibility in data representation, it also introduced increased computational complexity and the potential for precision loss. In 2018, the Fast Fully



$\vec{a}$ = [1,2,3, ... , n] or [1.3,2.5, ... , n] or [0,1,0, ... , n]
poly = [x^(n-1) +31x^(n-2) + ... + 21]
λ
Parameter Setup
$\vec{a}$
Slot / Coefficient Encoding
μ
Encryption $(\vec{pk})$
ct
rescaling / bootstrapping
Computations $(\vec{ek})$
Decryption (sk)
Decoding
$\vec{a}$
μ
poly = [x^(n-1) + 24565x^(n-2) + ... + 54]
ResNet-20
Convolutions
Pooling
Convolutions
Pooling
Fully Connected
Convolution
Input
Kernel Weights
32
32
X
pack into ciphertext
Input Ciphertext
perform (k x k = 5 x 5) rotation to access input ciphertext or convolution
rot 1
rot -1
rot 32
perform the remaining rotations for the ciphertext and multiply the weight plaintext
rotation keys required
symmetric pattern: [-66 -65 -64 -63 -62 -34 -33 -32 -31 -30 -2 -1 0 1 2 30 31 32 33 34 62 63 64 65 66]

Fig. 1: Overview of the computational processes for an FHE-protected application

Homomorphic Encryption Library (TFHE) [17] scheme was introduced, allowing bitwise operations by leveraging the ”gates” approach. Several open-source libraries [13], [17]–[20] have also been developed to enhance the accessibility of FHE. A prominent area of interest is the application of FHE to deep learning applications, particularly affected by the increasing popularity of cloud services. As a result, research efforts directed at overcoming the performance challenges associated with the implementation of FHE in deep learning, especially for privacy preservation, have become an active and dynamic field of study [2], [4]–[9], [11], [21].

B. Challenges of Memory and Performance Trade-Offs in Current FHE-based Privacy-Preserving System

Low-latency FHE-based neural network is becoming popular for applications that focused on in processing one or a few images [21]. Many of these FHE optimization techniques often trade-off memory to improve the time performance. For instance, [21] proposed a novel data packing method that organizes the data channel-wise, packing each channel into a single ciphertext, consuming $\approx$ 126 GB of memory for a 7-layer CNN. Following that, Kim et al. [5] introduced the Hyphen method, which tailored packing strategies (2D gap packing and the PRCR scheme) to enhance convolution efficiency. This approach reduces the latency of ResNet-20 on CIFAR-10 to 1.40 seconds on GPU, consuming up to 380 GB. Subsequently, Souhail et al. [6] optimized privacy-preserving deep learning by utilizing several techniques, including quantization and depth-optimized compressor-based accumulators. Nevertheless, their approach still necessitates 192 GB of memory to operate ResNet-20 on CIFAR-10. Similarly, Kim et al. [4] developed ConvFHE that leverages the properties of coefficient encoding to eliminate the need for rotation during the convolution. This approach delivers constant convolution time regardless of the kernel size, resulting in up to a $46\times$ speedup across different kernel sizes. It achieved 255 seconds per image inference on ResNet-20, attaining state-of-the-art accuracy of 92.02%. Similar to the previous works, it consumes a lot of memory (≈ 100 GB).

These research underscores a persistent trend in which optimizing execution time for FHE-based neural networks typically requires significant memory trade-offs. Accelerating models like ResNet-20 on CIFAR-10 demands up to hundreds of gigabytes of memory, a limitation that becomes more pronounced for larger, more advanced networks. De Castro et al. [7] also emphasize memory as a critical bottleneck in FHE applications, arguing that the performance gains cannot rely solely on computational acceleration if the inherent memory constraints in FHE are not addressed. This perspective is echoed by studies such as [8], [9], which note that the large memory footprint of rotation keys has increasingly become a limiting factor in the development of advanced, and efficient FHE-based privacy-preserving systems. Substantial memory requirements of current FHE-based neural networks present significant challenges to the deployment in industrial applications, particularly in scenarios involving outsourced services that require communication between users and servers. The primary concerns include:

1) **High Computational Overheads**: As system complexity increases, homomorphic computations require more rotations, resulting an extensive rotation key set that must be stored and transmitted. For example, [9] employed the technique in [3] to implement the CKKS scheme with a polynomial modulus degree $N = 2^{16}$ for a ResNet-20 on CIFAR-10, requires at least 265 rotation keys and consumed 105.60 GB of memory. [9] reports that generating this key set requires approximately 13 minutes on AMD Ryzen Threadripper PRO 3995WX CPU, underscoring the inefficiency when users need various rotation key sets for different services, which is a long waiting process. Thus, it is desirable to reduce the runtime and memory for rotation key generation to enhance usability.
2) **Inefficient Memory Management**: When accommodating large user bases, servers must allocate substantial memory for storing rotation keys per user. For example, supporting 1000 users with 512 GB of rotation key storage per user would necessitate 512 TB of memory. However, the memory allocated for keys that are not frequently accessed can lead to a waste of resources, while regenerating and retransmitting keys as needed also introduces overhead.

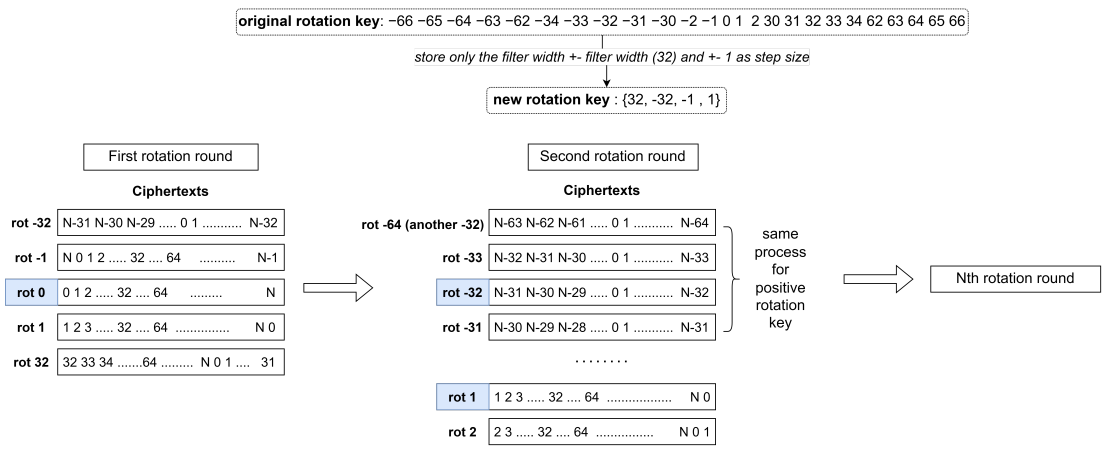

original rotation key: −66 −65 −64 −63 −62 −34 −33 −32 −31 −30 −2 −1 0 1 2 30 31 32 33 34 62 63 64 65 66
store only the filter width +- filter width (32) and +- 1 as step size
new rotation key : {32, -32, -1 , 1}
First rotation round
Ciphertexts
rot -32
N-31 N-30 N-29 ..... 0 1 .... N-32
rot -1
N 0 1 2 ..... 32 .... 64 .... N-1
rot 0
0 1 2 ..... 32 .... 64 .... N
rot 1
1 2 3 ..... 32 .... 64 .... N 0
rot 32
32 33 34 ....64 .... N 0 1 31
Second rotation round
Ciphertexts
rot -64 (another -32)
N-63 N-62 N-61 ..... 0 1 .... N-64
rot -33
N-32 N-31 N-30 ..... 0 1 .... N-33
rot -32
N-31 N-30 N-29 ..... 0 1 .... N-32
rot -31
N-30 N-29 N-28 ..... 0 1 .... N-31
same process for positive rotation key
rot 1
1 2 3 ..... 32 .... 64 .... N 0
rot 2
2 3 ..... 32 .... 64 .... N 0 1
Nth rotation round

Fig. 2: Example of rotation key and processes involved in LowMem technique [11]

C. Related Work

Recent research has switched their focus on optimizing the use of rotation keys to reduce memory consumption while maintaining efficient performance. For example, [9] introduces a hierarchical rotation key approach, which derives master keys and generates additional keys as needed, reducing the memory footprint for ResNet-20 on CIFAR-10 to about 2.91 GB. However, this method requires multiple rotation cycles to achieve the desired shifts, potentially increasing operation time and introducing more noise to ciphertext. The study primarily reports on key generation time, but does not investigate the trade-off between memory efficiency and potential degradation in inference times, raising questions about its feasibility in terms of speed performance. Moreover, Roshan et al. [8] observed that HEAAN [13] optimizes memory efficiency by employing powers-of-two to store rotation keys, requiring only $2\log(N) - 2$ rotation keys. However, they also pointed out that these rotation keys may not consistently align with the anticipated order in FHE, leading to inadequate coverage of the necessary rotation.

Rovida and Leporati [11] proposed an optimized approach by storing only the filter width and step size, instead of the entire set of rotation keys. It employed slot encoding in their implementation of ResNet-20 on CIFAR-10 dataset. Fig. 2 illustrates the sequence of rotation keys identified in [11] along with their corresponding rotation operations. For a 5×5 kernel size, the filter width is identified as 32, with each group positioned ± 32 indices apart. Additionally, a step size of ± 1 is consistently employed in their solution. Consequently, this approach allowed them to reduce the storage requirements from 25 rotation keys to just four key values: -32, 32, 1, and -1. Thus, the ciphertext at rotation indices of -1, 1, -32, and 32 can be obtained in the initial round using the original ciphertext at index 0 ($rot0$). These ciphertexts are then used in subsequent rotations to generate additional ciphertexts. Specifically, to produce the ciphertext at index -33, the rotated ciphertext at index -32 is subsequently rotated by -1. Likewise, to achieve the ciphertext at index -64, it is further rotated by -32. This process requires multiple rounds of group rotations to complete the overall ciphertext rotation operation. Although this technique [11] enhances memory efficiency without increasing the rotation count, it still encounters limitations that restrict its broader potential in memory reduction:

1) **Limited Support for Different Filter Sizes**: This approach exclusively supports odd-numbered filters and cannot accommodate rotation keys for even-numbered filters, such as 2×2 or 4×4.
2) **Assumption of Symmetric Patterns**: It assumes a symmetric numerical pattern for the rotation key. However, in practice, rotation keys often exhibit diverse patterns due to the unique architecture in different neural networks designed for FHE, which tends to be random for a large neural network.
3) **Inflexibility in Filter Width and Step Size Support**: The dependence on storing only the filter width and minimal step size presents challenges when the rotation keys do not align with a uniform step size or filter dimension.

As a result, this work [11] fails to address more complex scenarios such as [4], [8]. These investigations highlight the research gap regarding the optimization of rotation keys for memory management.

III. ARK

In this paper, the Adaptive Rotation Key (ARK) optimization technique is proposed to improve memory efficiency in FHE setups by minimizing the number of rotation keys. This technique aims to establish an optimal trade-off between processing time and memory consumption, thereby facilitating a memory-efficient solution for FHE-based neural networks. Fig. 3 illustrates the complete workflow of the ARK technique, which comprises three major steps:

1) **Identify Numerical Patterns and Generating Corresponding Pairs of Numbers**: This step involves identifying numerical patterns for rotation keys through Algorithm 1 (*Find Symmetric Pattern*), and Algorithm 2 (*Find Geometric Pattern*) to generate their respective mappings.

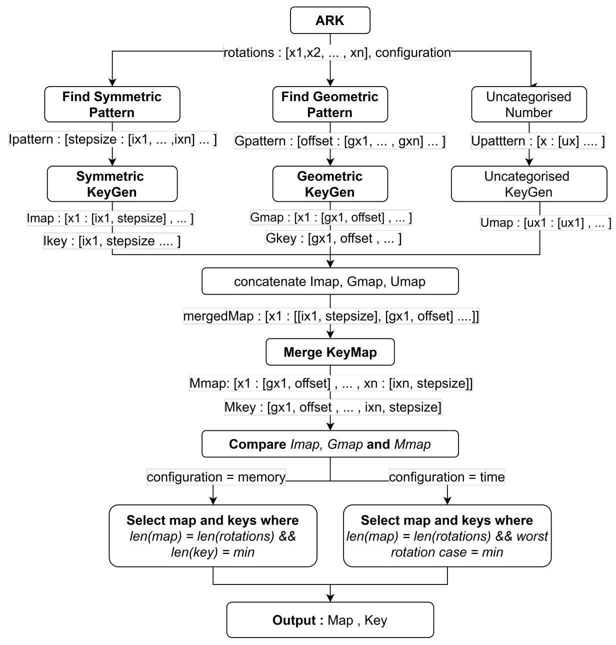

ARK
rotations : [x1,x2, ... , xn], configuration
Find Symmetric Pattern
Find Geometric Pattern
Uncategorised Number
Ipattern : [stepsize : [ix1, ... ,ixn] ...]
Gpattern : [offset : [gx1, ... , gxn] ...]
Upatttern : [x : [ux] ....]
Symmetric KeyGen
Geometric KeyGen
Uncategorised KeyGen
Imap : [x1 : [ix1, stepsize] , ...]
Ikey : [ix1, stepsize ....]
Gmap : [x1 : [gx1, offset] , ...]
Gkey : [gx1, offset , ...]
Umap : [ux1 : [ux1] , ...]
concatenate Imap, Gmap, Umap
mergedMap : [x1 : [[ix1, stepsize], [gx1, offset] ....]]
Merge KeyMap
Mmap: [x1 : [gx1, offset] , ... , xn : [ixn, stepsize]]
Mkey : [gx1, offset , ... , ixn, stepsize]
Compare Imap, Gmap and Mmap
configuration = memory
configuration = time
Select map and keys where len(map) = len(rotations) && len(key) = min
Select map and keys where len(map) = len(rotations) && worst rotation case = min
Output : Map , Key

Fig. 3: Process flow of the ARK technique

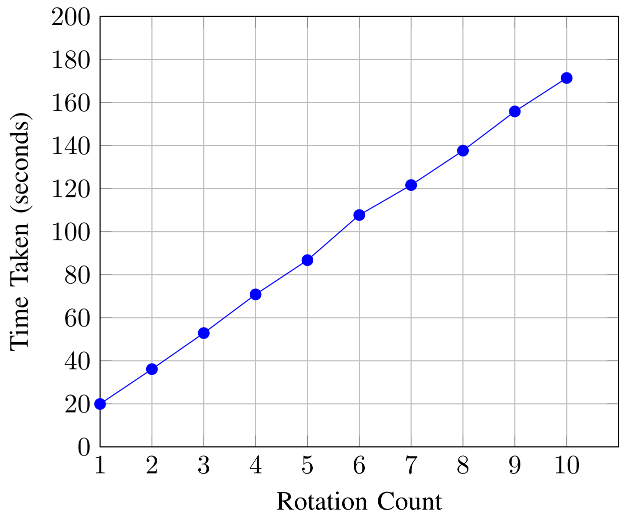


Fig. 4: The relationship between the number of rotation counts and the time taken to reconstruct a rotation number

2) **Creating Keys Based on the Filtered Pairs**: This step employs the identified mappings to generate the corresponding keys, utilizing *Symmetric KeyGen* (Algorithm 3) and *Geometric KeyGen* (Algorithm 4).
3) **Finalizing the Rotation Keys**: The final step focuses on producing a comprehensive set of minimal subset rotation keys that collectively cover all original rotation keys, through the *Merge KeyMap* (Algorithm 5).

Subsequently, ARK offers a dual-configuration, allowing users to select either a memory-optimized or time-optimized rotation key set. For maximum memory efficiency, the ARK technique selects the key set with the fewest rotation keys, minimizing memory usage. For optimized speed, it selects the map that minimizes the worst-case scenario of rotations when ciphertext pre-rotation criteria are not feasible. Ultimately, the ARK technique returns a minimal subset of rotation keys and a comprehensive map that aligns each original rotation key with its optimized counterpart. These keys are then applied to FHE parameters, while the maps are used to efficiently retrieve the derived rotations set. In summary, the ARK technique can accommodate rotation key sets with a variety of numerical patterns, covering most of the commonly found patterns encountered in FHE.

A. ARK formulation

Fig. 4 analyzes the relationship between rotation count and the time required for rotation key reconstruction in ConvFHE [4]. This underscores the trade-offs inherent in the system: higher rotation counts offer enhanced flexibility for key reuse and improved memory efficiency, yet it also introduces considerable processing overhead. Beyond a rotation count of two, the diminishing returns on time performance render further increases less advantageous. This aligns with findings in [3], suggesting that minimizing the rotation count is advantageous for reducing time. Rotation count of two is generally regarded as an optimal compromise, optimizing memory utilization with acceptable time performance.

To formalize this approach, a minimal subset $D \subset \mathbb{Z}$ is derived from a given set of rotation keys $S = \{s_1, s_2, \ldots, s_n\}$ such that every key $s_i \in S$ can be reconstructed using keys from D. The subset D may consist of integers that are either part of S or newly introduced numbers that maximize the potential for reuse when reconstructing its elements. This addresses the challenge of minimizing the cardinality of D, denoted as m, where $m < n$. It is to reduce the memory requirements for rotation keys while ensuring that each key s_i can be expressed as:

$$s_i = \begin{cases} d_j & \text{if } d_j \in D, \\ d_k + d_h & \text{if } d_k + d_h \in D, \end{cases} \qquad (1)$$
$$\forall\, j, k, h \in \{1, 2, \ldots, m\}.$$

To achieve an optimized solution, the following requirements are enforced: (1) the length of D must be smaller than that of S; (2) each key in D should be reused as much as possible to enhance the memory efficiency; and (3) each rotation key s_i may be constructed using either one or two elements from D to control the computational trade-offs associated with rotation operations in FHE.

B. Identify numerical pattern and generate mapping

The initial step in the ARK technique involves accepting a list of numbers and systematically identifying their relationships, categorizing them into specific patterns. The pattern of rotation keys is influenced by several factors:

1) How ciphertext transitions in a specific FHE scheme are managed between neural network layers that require rotation for appropriate data formatting.
2) The data packing and encoding method employed in the FHE operations also leads to variations in rotation key numerical pattern.

Algorithm 1 Find Symmetric Pattern

```
Require: A list of integers, $nums$
Ensure: A map of differences, $map : [stepsize : [v_1, \dots, v_n]]$

 1: $n \leftarrow$ sort($nums$)
 2: while $i \leftarrow 0$ to $i <$ len($n$) $- 1$ do
 3:     $start \leftarrow i$
 4:     $dif \leftarrow n[i+1] - n[i]$
 5:     while $i <$ len($n$) $- 1$ and $n[i+1] - n[i] = dif$ do
 6:         $i$++
 7:     end while
 8:     if $i > start$ and $(i - start + 1) > 1$ then
 9:         Append $n[start : i+1]$ to $patterns\ [dif]$
10:     end if
11:     $i$++
12: end while
13: return $patterns$
```

Upon investigation, two rotation key patterns have been identified in FHE-based neural networks. One prominent numerical pattern is the symmetric rotation key sequence as shown in Fig. 1, employing 5×5 filters with 32 filter width, characterized by 25 rotations grouped into sets of five with a step size of ± 1 that found in [4], [11] utilizes the slot encoding method. These rotations facilitate convolution operations and the transformation of ciphertext across neural network layers. However, the LowMem method [11] may encounter limitations when the rotation key does not adhere to a uniform step size. This is because such pattern is usually randomized in complex architectures [4], [8]. To extends beyond symmetric patterns, this work presents a novel approach that reduces rotation keys by identifying the central number and step size within each group, rather than merely storing a single filter and step size.

Algorithm 1 is designed to identify and group sequences of numbers that exhibit a consistent difference between consecutive elements in a list. The process begins by sorting the input integers to facilitate pattern detection (line 1). It then iterates through the sorted list and computes the difference between consecutive numbers (lines 2 to 4). When a uniform difference is discovered across a subset of elements, they are stored in a data map that associates the difference with the list of numbers forming that sequence (e.g., $map\ [\ stepsize : [value_1,\ value_2,\ \dots\ ,\ value_n]]$). The algorithm ensures that only sequences comprising more than two numbers are retained (lines 5 to 12), emphasizing the significance of sufficiently long patterns for effective memory optimization. This process repeats until all elements are analyzed, then output a collection of symmetrical patterns, where each key represents a specific difference, and its associated values are the integers that follow that arithmetic progression. The sample output is illustrated as the $categorisedNumber$ in Fig. 5. This approach overcomes the limitations present in prior work [11] by providing a concise representation of the underlying numerical structure, resulting a more versatile solution applicable to a wide range of contexts.

In addition, geometric patterns are also prevalent in the rotation key sequences, particularly when employing coefficient encoding [4]. In this context, each number is derived by multiplying the preceding term by a constant ratio. This limitation renders the LowMem technique [11] unsuitable for such cases. Consequently, the second aspect of ARK focuses on identifying and categorizing geometric patterns within the rotation key list. It has been observed that the rotation keys frequently adhere to a pattern related to powers-of-two; yet, they do not always align precisely with the values of the power-of-two sequences. For instance, one specific list in ConvFHE [4] includes the following values: $3, 5, 9, 17, 33, 65, 129, 257, 513, 1025, 2049, 4097, 8193, 16385, 32769, 65537$. Each of these values can be expressed as $2^n + 1$, where $n \leq 16$. These numbers can be categorized following the equation: $2^n \pm k$, where k serves as an offset to adjust the number to its actual value.

Algorithm 2 Find Geometric Pattern

```
Require: A list of integers, $nums$
Ensure: A map of differences, $map : [k : [v_1, v_2, \dots, v_n]]$
 1: for each $a$ in $nums$ do
 2:     for $n \leftarrow 0$ to $\lfloor \log_2(a) \rfloor + 1$ do
 3:         $base \leftarrow 2^n$
 4:         $k \leftarrow a - base$
 5:         if $k \notin kGroups$ then
 6:             $kGroups[k] \leftarrow a$
 7:         else if $a \notin kGroups[k]$ then
 8:             Append $a$ to $kGroups[k]$
 9:         end if
10:     end for
11: end for
12: for each $k, group$ in $kGroups$ do
13:     if len($group$) $< 2$ then
14:         Remove $k$ from $kGroups$
15:     end if
16: end for
17: return $kGroups$
```

Algorithm 2 is developed to identify and group integers based on their geometric relationships within a specified list. For each number in the input list (*nums*), the algorithm calculates the potential bases derived from power-of-two (line 3). Subsequently, it computes the difference k between the number and the nearest identified base (line 4) and group the number under k into *kGroups* map (lines 5 to 9). Any groups containing fewer than two numbers are removed (lines 12 to 16), because they do not form valid geometric sequences; directly assigning the original rotation key is deemed more memory efficient. The algorithm produces a structured map where the keys represent the constant differences k, and the associated values are rotation keys that share the same difference, organized as $map\ [\ k : [value_1,\ value_2,\ \dots,\ value_n\ ]]$. A sample output is illustrated as the *categorizedNumber* in Fig. 6. Finally, any numbers that remain uncategorized after the execution of Algorithms 1 and 2 are added directly to the map as original keys. All generated maps will be passed on to phase two of the algorithm, which is dedicated to key

Algorithm 3 Symmetric KeyGen

```
Require: A map of integer numbers, $nums$.
Ensure: A list of filtered keys, $map[int][]int$ and a map $int[]$ of key offsets.
1: for each $z, n$ in $nums$ do
2:     $curGrp \leftarrow n[0]$
3:     for $i \leftarrow 1$ to $i <$ len($n$), +=1 do
4:         $dif \leftarrow n[i] - n[i-1]$
5:         if $dif > z$ then
6:             $mid \leftarrow$ findCenter($curGrp$)
7:             $grps[mid] \leftarrow curGrp$, Append $mid$ to $centVal$
8:         else
9:             Append $n[i]$ to $curGrp$.
10:        end if
11:    end for
12: end for
13: for $z, n$ in $grps$ do
14:    if moreSubdivision($n$) then
15:        $mid \leftarrow$ findCenter[$n[s:e]$]
16:        $grps[mid] \leftarrow n[s:e]$, Append $mid$ into $centVal$
17:    end if
18: end for
19: for each $mid$ in $centVal$ do
20:    for each $val$ in $grps[mid]$ do
21:        $k \leftarrow val - mid$
22:        $pm[val] \leftarrow []int\{mid, k\}$, Append $k$ to $pk$
23:    end for
24: end for
25: $pk \leftarrow$ concat($pk, centVal$)
26: for each $ky$ in $pm$ do
27:    if $ky$ exist in $pk$ then
28:        $pm[ky] \leftarrow ky$
29:    end if
30: end for
31: return $pk$ and $pm$
```

generation.

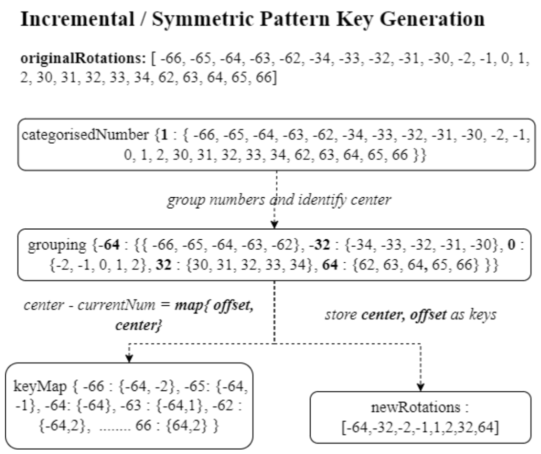

Incremental / Symmetric Pattern Key Generation
originalRotations: [-66, -65, -64, -63, -62, -34, -33, -32, -31, -30, -2, -1, 0, 1, 2, 30, 31, 32, 33, 34, 62, 63, 64, 65, 66]
categorisedNumber {1 : { -66, -65, -64, -63, -62, -34, -33, -32, -31, -30, -2, -1, 0, 1, 2, 30, 31, 32, 33, 34, 62, 63, 64, 65, 66 }}
group numbers and identify center
grouping {-64 : {{ -66, -65, -64, -63, -62}, -32 : {-34, -33, -32, -31, -30}, 0 : {-2, -1, 0, 1, 2}, 32 : {30, 31, 32, 33, 34}, 64 : {62, 63, 64, 65, 66} }}
center - currentNum = map{ offset, center}
store center, offset as keys
keyMap { -66 : {-64, -2}, -65: {-64, -1}, -64: {-64}, -63 : {-64,1}, -62 : {-64,2}, ... 66 : {64,2} }
newRotations : [-64,-32,-2,-1,1,2,32,64]

Fig. 5: Example process of number filtering and key generation for incremental/symmetrical numerical pattern.

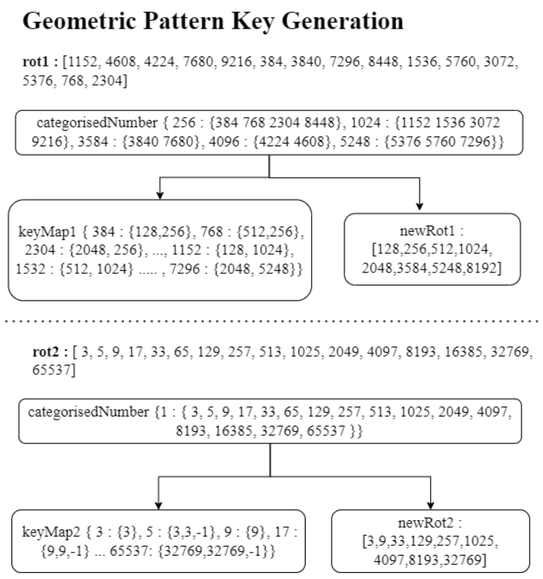

Geometric Pattern Key Generation
rot1 : [1152, 4608, 4224, 7680, 9216, 384, 3840, 7296, 8448, 1536, 5760, 3072, 5376, 768, 2304]
categorisedNumber { 256 : {384 768 2304 8448}, 1024 : {1152 1536 3072 9216}, 3584 : {3840 7680}, 4096 : {4224 4608}, 5248 : {5376 5760 7296}}
keyMap1 { 384 : {128,256}, 768 : {512,256}, 2304 : {2048, 256}, ..., 1152 : {128, 1024}, 1532 : {512, 1024} ..... , 7296 : {2048, 5248}}
newRot1 : [128,256,512,1024, 2048,3584,5248,8192]
rot2 : [3, 5, 9, 17, 33, 65, 129, 257, 513, 1025, 2049, 4097, 8193, 16385, 32769, 65537]
categorisedNumber {1 : { 3, 5, 9, 17, 33, 65, 129, 257, 513, 1025, 2049, 4097, 8193, 16385, 32769, 65537 }}
keyMap2 { 3 : {3}, 5 : {3,3,-1}, 9 : {9}, 17 : {9,9,-1} ... 65537: {32769,32769,-1}}
newRot2 : [3,9,33,129,257,1025, 4097,8193,32769]

Fig. 6: Example process of number filtering and key generation for geometric numerical pattern

C. Rotation Key Generation

In this phase, the aim is to process the previously filtered maps to facilitate key generation. Algorithm 3 utilizes the symmetrical map to generate an optimized rotation key set for each identified rotation key, along with the corresponding mapping keys in subset D, thereby facilitating the reconstruction of the original key. The algorithm begins forming initial groups by iterating through the input map. It initializes a *curGrp* with the first number, then examines the differences between consecutive numbers in the sorted list (lines 1 to 4). Whenever the difference exceeds the discovered step size (z), a new subgroup is created. The center of each *curGrp* is calculated and stored in the group map, while the center values are added to the *centVal* list, which is part of the minimal subset key (lines 5 to 9). This ensures that each group contains numbers that are closely related based on their differences. In certain cases, the identified patterns may lack distinct filter sizes. For example, assuming there is a rotation key set containing values from 0 to 10 would result in all numbers within this range being classified as a single group. In this scenario, the algorithm assigns 5 as the center value, with step sizes extending from -4 to 5. Consequently, this limits the ability to optimize rotation key memory by retaining ten rotation keys: one for the center value and nine for the associated step sizes. This situation underscores the challenge of efficiently managing rotation key storage when symmetrical patterns exhibit uniformity across a large group of numbers. To overcome this, the algorithm further divides the grouped number into smaller, more manageable subgroups. It employs a factor-based approach to distribute numbers into smaller groups whenever feasible. The centers of these subgroups are calculated and stored in the groups map, with their respective center values added to *centVal* list (lines 13 to 17). If a group cannot be divided, it is retained in its original form to preserve its structure.

Following that, offset rotation keys are calculated for each number within the processed groups. Each offset represents the difference between the current number and its designated group center. The *pm* map is then updated to include this offset, along with the centers, as part of the newly optimized rotation key set (lines 19 to 24). Consider a subgroup of numbers like $66, 65, 64, 63$, and 62, the center value is 64, and the associated step sizes are ±1 and ±2. It is sufficient to retain only the center value for each group, enabling a diverse step size to be uniformly applied across various symmetric subgroups. The final rotation keys are produced by merging the center values with all identified offset values, ensuring that duplicates are eliminated to maintain uniqueness (line 25). Finally, the algorithm filters out unnecessary entries, particularly if their corresponding direct keys exist in the new rotation list (lines 26 to 30). This step significantly reduces redundancy, ensuring that the resulting list of keys is concise and efficient. Fig. 5 summarizes the entire process of number filtering and key generation for incremental and symmetric pattern rotation keys.

Performing two times of homomorphic rotation for a specific number can be interpreted as an addition operation. The geometric rotation key pattern with common factor characteristic requires careful formulation to balance the trade-off between execution time (the number of sub-rotations required) and memory optimization. Consider a list of numbers exhibiting a geometric pattern: $2, 4, 8, 16, 32, 64$, and 128, where each number is a multiple from its predecessor. Employing the symmetrical algorithm approach would lead to an increased number of rotation counts for the larger numbers. For instance, the base number (2) is stored in this sequence, obtaining eight necessitates four rotations {2, 2, 2, 2}, resulting in significant performance degradation. To enhance optimization, the focus shifts toward numbers that are powers-of-two, thereby minimizing memory usage. The $rot1$ list as depicted in Fig. 6, is derived from ConvFHE [4] experiments. The numbers are categorized by the offset, k, with each key set adhering to the formula $2^n + k$. For example, the first key with a value of 256, the following power-of-two values is obtained:

$$384 - 256 = 128 \quad \text{(which is } 2^7\text{)},$$

$$768 - 256 = 512 \quad \text{(which is } 2^9\text{)}.$$

Thus, the new rotation key subset should consist of one offset and one value corresponding to a power-of-two. In the case of the offset 1024, obtaining 1152 involves $1024, 128$, which shares the same power-of-two with 384. Consequently, the number of rotations in the initial configuration for $rot1$ realizes an approximate 46% reduction in key storage.

The rotation key may consist of only power-of-two numbers that cannot be effectively categorized using the factorization method, as illustrated by $rot2$ in Fig. 6, where by only one mapping that encompasses the entire number set exists. Without optimization, the algorithm retains the entire key set from 2^1 to 2^{15}, leading to inefficiencies. Hence, it is proposed to store rotation values in groups of two, using a stridden approach while flipping the signs of the offset keys, such as $-1, 3, 9, 33$, and so on. This method enables the utilization of the first stride number, such as 3, to reconstruct subsequent values like 5 through the combination $\{3, 3, -1\}$. However, the same base number (3) cannot be employed to construct higher numbers, such as 17, which would require $6 \times 3 - 1$. This results in excessive rotation cycles that could significantly degrade performance. By adopting a stridden method where 3 constructs 5, 9 constructs 17, and 33 constructs 65, and utilizing flipped offset keys, a more optimal solution for this scenario is achieved.

Algorithm 4 Geometric KeyGen

```
Require: A map of integer numbers, $nums$.
Ensure: A list of filtered keys $int[]$, and $map[int][]int$ a map
    of key offsets.
 1: for each $ky$, $num$ in $nums$ do
 2:   $v \leftarrow num$
 3:   if ConsecutiveNum($v$, $ky$) and len($v$) $> 4$ then
 4:     Append $-ky$ to $pk$
 5:     for $i \leftarrow 1$ to len($v$), +=2 do
 6:       $pm[v[i-1]] \leftarrow v[i-1]$, Append $v[i-1]$ to $pk$
 7:       $pm[v[i]] \leftarrow []int\ \{v[i-1], v[i-1], -ky\}$
 8:     end for
 9:   else
10:     for each $a$ in $v$ do
11:       if isPowerOfTwo($ky$) then
12:         $pm[a] \leftarrow []int\ \{a - ky, ky\}$, Append $ky$ and
            $a - ky$ to $pk$
13:       else if $ky = 0$ then
14:         $pm[a] \leftarrow a$, Append $a$ to $pk$
15:       else
16:         $rm \leftarrow nums[ky]$
17:       end if
18:     end for
19:   end if
20: end for
21: Sort $rm$ in descending order
22: for each $ky, v$ in $rm$ do
23:   if $val$ not exists in $pm$ then
24:     $tm[v] \leftarrow []int\{v - ky, ky\}$
25:   end if
26: end for
27: for each $ky, combo$ in $tm$ do
28:   if all values in $combo$ exists in $pk$ then
29:     $pm[ky] \leftarrow combo$, Append $combo$ into $pk$
30:   else if either 1 value in $combo$ exist in $pk$ then
31:     $pm[ky] \leftarrow combo$, Append $combo$ into $pk$
32:   end if
33: end for
34: return $pk$ and $pm$
```

Algorithm 4 is developed to generate optimized rotation keys for geometric numerical patterns, with a specific focus on identifying the most effective combinations to express target numbers while minimizing redundant calculations. The algorithm begins by iterating through the input map to assess if the geometric pattern aligns with the criteria for consecutive numbers in the list, akin to the $rot2$ pattern. If it meets this criterion, the rotation key is stored in a stridden format, with the keymap stored in pm and the keys stored in pk for retrieval (lines 3 to 9). Conversely, if the grouping in the input map appears random, like $rot1$, the algorithm prioritizes key mappings where $k = 0$. This corresponds to the exact power-of-two values and combinations where either one value is the exact power-of-two value (lines 10 to 15). These values are decomposed into pairs, identifying the nearest power-of-two alongside its corresponding offset (e.g., $rot : \{k, 2^n\}$). Any remaining rotations that do not yield an offset combination are transferred to rm for further processing (line 16).

Following this, the algorithm prioritizes keys from rm based on the length of values associated with each key, sorting them in descending order (line 21). Upon establishing this order, unselected rotations from rm are incorporated into a temporary map (tm) to prioritize the combinations that cover the most numbers (lines 22 to 26). This strategy ensures that the keys with a broader coverage are processed first. At this stage, each unselected rotation may belong to multiple key groups in tm. Subsequent iteration examines tm and selects the most efficient combination for each rotation (lines 27 to 33), and produces the final rotation keys (pk) and rotation map (pm), representing the most optimized strategy for geometric key generation. In summary, the geometric key generation is guided by the following criteria:

1) Prioritization of key sets belonging to the power-of-two group.
2) Prioritization of key sets where both values in the combination exist in the global key list. For example, if 7 has the combination {3, 4}, and both 3 and 4 are already present as keys in pk, this combination is prioritized.
3) If the above criteria are not met, select key sets where at least one value in their combination exists in the global key list.
4) If none of the previous criteria are fulfilled, direct assignment of the number as a key.

D. Finalizing Rotation Key

Upon completing the second phase of key generation for each numerical pattern list, two distinct rotation key lists were obtained, each accompanied by their respective keymaps. Nevertheless, these generated keymaps do not inherently guarantee full coverage of the rotations from the original set. This is because some values may be associated with only one numerical pattern. To address this limitation, Algorithm 5 integrates the key combinations derived from both the symmetric and geometric key sets. This integration aims to produce a comprehensive key map that ensures full coverage of all values in the original rotation key list, while simultaneously selecting the most optimal combination to achieve this objective.

Initially, this algorithm prioritizes selection of rotation keys that are associated with single-group values (lines 1 to 5). Next, the algorithm processes the remaining keys by iterating through all possible combinations for each rotation, specifically selecting those combinations where all values already exist in the final key list to ensure efficient rotations (lines 6 to 12). A brute-force approach is employed to assess the remaining combinations to select the one that requires the fewest unique additions to the final key list (lines 13 to 23), thereby minimizing the number of rotation keys introduced into the final key list. In this work, greedy algorithms and dynamic programming filtering methods were also evaluated to determine the optimal subset of keys. Nevertheless, the brute-force approach was selected due to its ability to consistently generate the most optimal solution compared to the other methods, and swift completion time in a few seconds.

Algorithm 5 Merge KeyMap

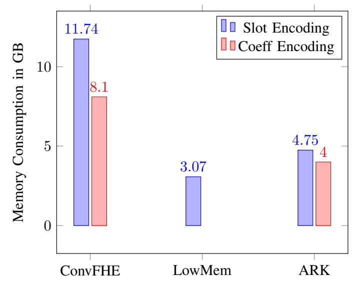
```
Require: A map of rotation keys compiled from multiple merged maps, $inputMap$
Ensure: A list of filtered keys $int[]$, and $map[int][]int$ a map of key offsets
 1: for each $k, v$ in $inputMap$ do
 2:     if len($v$) $= 1$ then
 3:         $map[k] \leftarrow v[0]$, Append $k$ into $keys$
 4:     end if
 5: end for
 6: for each $key, combinations$ in $inputMap$ do
 7:     for each _, $combo$ in $combinations$ do
 8:         if all values in $combo$ exists in $keys$ then
 9:             $map[key] \leftarrow combo$, Append $combo$ into $keys$
10:         end if
11:     end for
12: end for
13: for $key, combinations$ in $inputMap$ do
14:     $minUnique \leftarrow -1$
15:     for $combo$ in $combinations$ do
16:         $unique \leftarrow$ countUnique($combo, keys$)
17:         if $minUnique = -1 || unique < minUnique$ then
18:             $minUnique \leftarrow unique$
19:             $bestCombo \leftarrow combo$
20:         end if
21:     end for
22:     $map[key] \leftarrow bestCombo$, Append $key$ into $keys$
23: end for
24: return $keys$, $maps$
```

IV. Experimental results and discussions

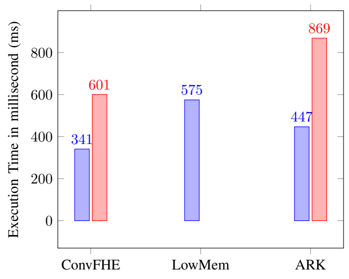
This section presents an experimental evaluation of the proposed ARK technique as applied to ConvFHE [4], utilizing CKKS FHE scheme and a ResNet-20 network. ConvFHE [4] explored a range of configurations across kernel sizes (3, 5, 7), layer depths (8, 14, 20), and wideness factors (1, 2, 3) for sparse data packing, collectively referred to as $K - L - W$ experiment (e.g., 3-20-1). In this paper, the original ConvFHE configuration is used as the baseline, while ARK is assessed under two configurations: Memory-Prioritized (MP) and Time-Prioritized (TP).

A. Performance Evaluation of Rotation Key Management in ResNet-20 for Single-Layer Convolution

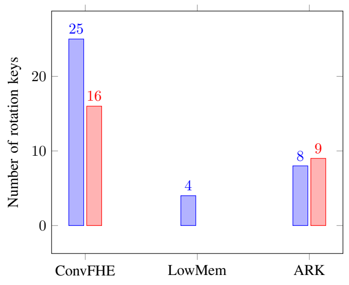
Fig. 7 presents a comparison of memory usage, execution time, and rotation key count for single-layer convolution for



(a) Memory



(b) Time



(c) Rotation Key

Fig. 7: Comparison of memory, time, and rotation keys for slot encoding and coefficient encoding in single-layer convolution with [4], [11]

TABLE I: Impact of rotation keys on total memory usage in ConvFHE baseline [4] experiments

K-L-W	**3 -20-1**	**5-20-1**	**7-20-1**	**5-8-3**	**3-14-3**	**3-20-3**
Rotation Key Memory (GB)	81.37	81.05	81.78	88.31	88.75	88.85
Total Memory (GB)	86.74	87.56	87.88	93.05	92.39	90.39
Rotation Key Contribution to Total Memory (%)	93.81	94.85	93.06	94.91	96.06	98.30

Note: ***K-L-W*** *denotes Kernel Size, Layer, and Wideness Factor for each experiment.*

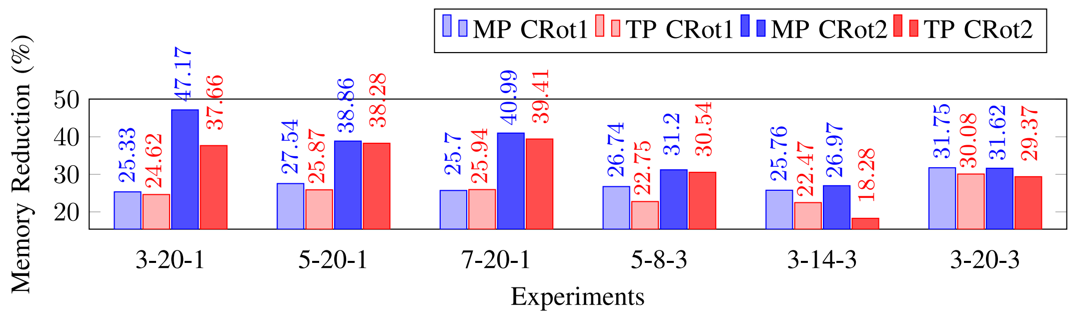
slot and coefficient encoding methods. For slot encoding, the ConvFHE (baseline) implementation necessitates 25 rotation keys, consuming 11.74 GB (Fig. 7a). In contrast, the proposed ARK technique reduces the required rotation keys to 8 (Fig. 7c), achieving 68.00% reduction in key usage and a 59.54% memory reduction, resulting in only 4.75 GB (Fig. 7a) of memory consumption. The LowMem [11] method can further reduces rotation keys to just 4 (Fig. 7c), which consumes only 3.07 GB, which reduces more memory compared to ARK. This is because LowMem [11] only retains the filter width offsets (± 32) and a constant step size (± 1), so the rotation key count is constant (always four). However, this approach cannot be used when the subgroup intervals are inconsistent, while the proposed ARK technique can accommodate variable subgroup intervals by categorizing values into groups. For instance, in the case of a 5×5 filter Fig. 2, LowMem [11] requires only the keys $32, -32, 1, -1$, while ARK retains $-64, 64, -32, 32, -1, 1, -2, 2$ to accommodate varying intervals, resulting in slightly higher memory usage.

Note that such compromise is particularly significant in scenarios where the rotation key sequence becomes more complex than a symmetric pattern, especially for large neural network FHE sequence where the often exhibit randomness. Although all methods evaluated are having the same rotation count throughout the experiments, LowMem [11] exhibits higher latency compared to the ARK technique, with latencies of 575 ms for LowMem [11] and 447 ms for ARK (Fig. 7b). This discrepancy in latency is primarily due to the greater coverage of rotation keys in the ARK, which reduces the overhead associated with initiating rotation operations. Notably, the LowMem [11] requires multiple rotations to complete (see Fig 2), but the proposed ARK technique benefits from fewer iterations due to its larger number of stored rotation keys. Thus, ARK operates 22.26% faster than the LowMem [11]. This efficiency leads to a lower latency increase in ARK (31.09%) compared to ConvFHE baseline [4], whereas the LowMem [11] is significantly slower (68.62%) than [4].

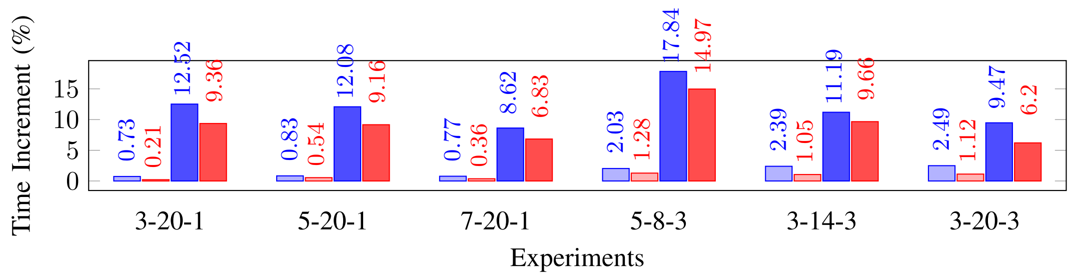
On the other hand, the LowMem [11] method assumes that the rotation key set is always symmetric, rendering its solution inapplicable for coefficient encoding convolution in [4]. ARK can effectively reduces the rotation key count from 16 to 9, achieving a 43.76% reduction in keys and contributing to a 50.62% decrease in memory usage, dropping from 8.10 GB to 4.00 GB. However, the time performance also degrades by 44.77%, increasing from 601 ms to 869 ms. Consequently, the performance of ARK for coefficient encoding is less effective than for slot encoding. This can be attributed to the inherent inconsistency of geometric numerical patterns compared to symmetric numerical patterns, where geometric sequences often arise from multiplicative factors rather than arithmetic differences. Such inconsistencies impose limits on key summarization, as the values increment in powers, thereby reducing the potential for shared key reuse. As a result, the overall rotation count required to complete the original rotation set tends to increase, as illustrated in Fig. 6 $rot2$ example, which demonstrates the increased need for additional rotations.

B. Impact of Rotation Key Memory on Total Memory Usage

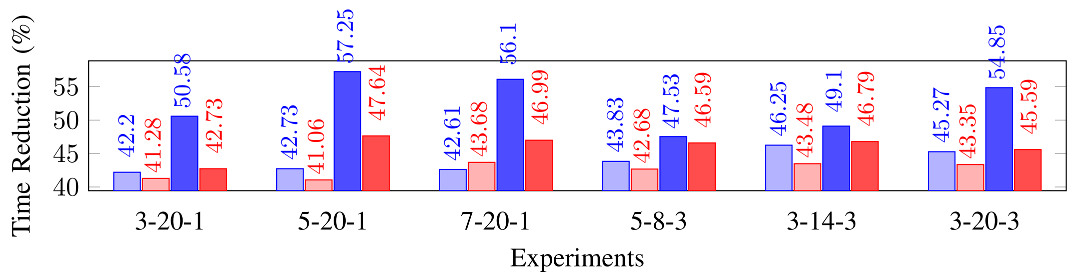
Table I presents the impact of rotation key memory on the total memory usage across various experiments in the ConvFHE baseline [4]. The results were obtained from the source code released by [4] in open source domain. The results indicate that the memory usage for experiments with a wideness factor of 1 ranges from 81.05 GB to 81.78 GB, while for a wideness factor of 3, it ranges from 88.31 GB to 88.85 GB. The higher memory consumption observed in the wideness factor of 3 can be attributed to the increased requirement for rotation keys due to the sparse packing of input data, resulting in greater overall memory usage compared to the wideness factor of 1. Furthermore, ConvFHE [4] demonstrates an overall memory consumption ranging from 86.74 GB to 93.05 GB. A detailed breakdown indicates that the contribution of rotation keys to total memory usage is substantial, accounting for approximately 93.06% to 98.30%. It highlights the potential



(a) Rotation Key Memory



(b) Execution Time



(c) Rotation Key Generation Time

Fig. 8: Analysis of rotation key memory reduction, execution time increment, and rotation key generation time reduction with ARK across various configurations and experiments

Notes: (1) ***MP*** *denotes memory prioritized configuration. (2)* ***TP*** *represents time prioritized configuration.*

for significant reduction through effective management. Thus, optimizing rotation key usage emerges as a vital strategy for advancing efficiency in ConvFHE [4] implementations.

C. Memory and Time Performance Analysis of ARK

Throughout the experiment, two distinct scenarios were discovered for implementing ARK in ConvFHE [4]. The first rotation scenario is applied during the block transition of the ciphertext in ResNet-20, referred as `CRot1`. This process iterates over the offset map, multiplying each offset with a single input ciphertext then rotating it to produce the result. Another rotation is employed during ciphertext packing after convolution, referred as `CRot2`. It accepts an array of inputs and offsets in ciphertext format, performs a few homomorphic operations, rotates it, and then reassigns the results back to their respective slot indices in the inputs for the next iteration. Table II presents the impact of ARK on rotation key memory and total memory usage across different rotation scenarios and configurations. ARK significantly reduces rotation key memory from an initial range of 81.05 GB to 88.85 GB to a reduced range of 47.87 GB to 70.99 GB, yielding rotation key memory reduction of approximately 18.28% to 47.17% (Fig. 8a). This improvement also reduces the overall memory usage up to 35.05% to a minimum of 56.34 GB. On the other hand, Table III analyzes the impact of ARK on execution time, showing an increase from ConvFHE baseline range of 272 to 594 seconds to 275 to 650 seconds, reflecting a time increase of around 0.21% to 17.84%, as illustrated in 8b. These findings indicate that the memory savings achieved in rotation key storage outweigh the associated time increase, supporting ARK as a beneficial trade-off technique for memory efficiency. `CRot1` and `CRot2` reveals significant differences in memory efficiency and execution time. `CRot1` demonstrates memory reduction in rotation key between 22.47% and 31.75% in rotation key memory (see Fig. 8a), while incurring a minimal time increase of 0.21% to 2.49% (Fig. 8b), compared to the ConvFHE baseline. In contrast, `CRot2` increases execution time ranging from 6.20% to 17.84% (Fig. 8b), to achieve

TABLE II: Comparison of Rotation Key Memory Reduction and Overall Memory Savings with ARK Across Different Configurations and Rotation Scenarios in ConvFHE

K-L-W			3-20-1	5-20-1	7-20-1	5-8-3	3-14-3	3-20-3
Memory Priortized Configuration (GB)	**Rotation Key**	**ConvFHE**	81.37	81.05	81.78	88.31	88.75	88.85
		CRot1	60.76	58.73	60.76	64.70	65.89	64.89
		CRot2	**47.87**	49.55	48.26	60.76	60.57	60.76
	Total	**ConvFHE**	86.74	87.56	87.88	93.05	92.39	90.39
		CRot1	62.61	61.94	62.61	67.55	68.75	68.75
		CRot2	**56.34**	59.54	59.28	62.61	62.12	62.61
Time Priortized Configuration (GB)	**Rotation Key**	**ConvFHE**	81.37	82.75	81.78	88.31	87.73	86.85
		CRot1	61.34	61.34	60.57	68.22	68.02	70.99
		CRot2	50.73	51.07	49.55	61.34	61.34	61.34
	Total	**ConvFHE**	86.74	87.86	87.88	93.05	92.39	90.39
		CRot1	63.89	63.89	62.12	71.07	70.88	71.85
		CRot2	60.04	60.46	59.54	63.89	63.89	63.89

TABLE III: Comparison of Rotation Key Generation Time and ResNet Execution Time with ARK Across Various Configurations and Rotation Scenarios in ConvFHE

K-L-W			3-20-1	5-20-1	7-20-1	5-8-3	3-14-3	3-20-3
Memory Priortized Configuration (seconds)	**Rotation Key Geneneration**	**ConvFHE**	495	498	499	525	530	530
		CRot1	274	271	276	290	276	275
		CRot2	232	**202**	205	271	261	225
	Resnet Execution	**ConvFHE**	406	405	429	272	436	594
		CRot1	409	409	432	277	447	607
		CRot2	457	454	466	320	485	650
Time Priortized Configurtion (seconds)	**Rotation Key Generation**	**ConvFHE**	495	497	510	525	530	530
		CRot1	278	280	274	296	290	293
		CRot2	273	248	257	276	273	275
	Resnet Execution	**ConvFHE**	406	404	429	272	436	594
		CRot1	407	406	430	**275**	441	601
		CRot2	444	441	458	312	478	631

memory saving between 18.26% and 47.17% (Fig. 8a). This discrepancy arises from the operational design of `CRot1`, wherein each offset value is independently applied to a consistent input. By leveraging ARK's keyMap to pre-rotate the input ciphertext into a series of pre-rotated ciphertexts, offsets can be adjusted accordingly in raw form to align with these pre-rotated inputs. The resulting pre-rotated ciphertext can then perform homomorphic operations with the offsets without increasing the rotation count. Thus, the best and worst-case scenarios for rotation in `CRot1` remain consistent with the original rotation count.

In contrast, `CRot2` presents a distinct challenge as it operates on an array of ciphertexts as inputs instead of one ciphertext as in `CRot1`. Pre-rotating the input ciphertext results in significant memory expansion, requiring storage for `rotation count` × `number of input ciphertexts`. Since the offsets are in ciphertext format, the memory required to align offset ciphertexts to pre-rotated ciphertexts may potentially doubled. However, achieving pre-rotation of ciphertext is impractical in `CRot2` due to the dependency of the input ciphertext on the operations executed within the loop. For instance, it utilizes an input ciphertext at specific slot indices, which must be computed, rotated, and then reassigned with the computed values to the corresponding slots. That means the input ciphertext used for operations is contingent upon the results from the previous round, rendering the pre-rotation of ciphertext unfeasible. Therefore, the rotation of input ciphertext must occur concurrently with the loop iterations rather than being conducted in advance. For instance, as illustrated in Fig. 6 $rot1$, the original rotation count is 15, ARK necessitates more than 15 rotations due to the use of a one to two subset rotation to replace a single rotation, resulting in increased rotation count and time latency. However, the memory reductions achieved outweigh the time increments, thereby presenting a favorable trade-off.

Ultimately, ARK offers users the flexibility to prioritize either maximum memory reduction or a balanced approach that allows for a certain amount of memory reduction while maintaining faster execution times. As shown in Fig. 8a, the lowest rotation key memory achieved through a memory-prioritized configuration ranges from 25.70% to 41.17%, which is notably higher than the time-prioritized configuration. The memory configuration averages a 7.40% greater rotation memory reduction compared to the time configuration. Conversely, the time configuration, as illustrated in Fig. 8b, incurs a lower execution time increase of 0.21% to 14.97% compared to the execution time increase of 0.73% to 17.84% in the memory configuration.

ARK can reduce rotation key count ranging from 26.18% to 48.71%. This decrease in rotation key count is associated with a significant reduction in rotation key generation time, ranges from 41.06% to 57.25%, as shown in Fig. 8c, corresponding to a duration of 205 to 296 seconds, as detailed in Table III. Notably, the operational overhead associated with the ARK technique is minimal, with a maximum time consumption of only 5 seconds. This reduction in rotation key generation time is particularly advantageous for server memory management, facilitating more efficient memory allocation by allowing for the release of rotation keys when they are not in use and enabling their regeneration as needed. In terms of accuracy, ARK exhibits a near-zero accuracy degradation ($\leq$ 0.03%) compared to the accuracy reported in ConvFHE [4].

Furthermore, ARK was applied to another research framework [22] that proposed a novel packing method, CBC, which significantly reduces the rotation operations required in FHE for ResNet-20. By implementing ARK, this rotation key was further reduced from 45 to 29 keys, consuming only 3.64 GB. However, we are unable to directly evaluate ARK's impact on CBC because the source code is not available publicly. Nevertheless, our analysis indicates that the rotations are conducted within a single ciphertext during convolution—a process closely aligned with the scenario addressed in `CRot1`. This similarity lends strong confidence that ARK can achieve a 47.32% reduction in memory with a manageable trade-off in time performance.

V. Conclusion

The proposed ARK technique systematically identifies optimized subsets of rotation keys to minimize storage demands. It also employs a dual configuration, allowing users to prioritize either memory or speed performance based on application needs. Experimental results show that implementation of ARK in ConvFHE [4] can achieve up to a 41.17% reduction in memory usage when memory optimization is prioritized. Alternatively, when prioritizing minimal time impact, ARK achieved a 22.75% memory reduction with a minor time increase of only 0.21%, approximately. This shows that ARK is capable in enhancing FHE's adaptability and performance across a variety of applications, underscoring its role in advancing practical solutions for privacy-preserving systems under diverse operational constraints.

References

[1] H. Narumanchi, N. Emmadi, and P. Gauravaram, "Costs of encrypted computation: Why fully homomorphic computations are slow," Tata Consultancy Services, 2020, [Online]. Available: https://www.researchgate.net/publication/354842186_Costs_of_Encrypted_Computation_Why_Fully_homomorphic_computations_are_Slow.

[2] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," *Proceedings of the 39th International Conference on Machine Learning*, vol. 162, pp. 12 403–12 422, 2022.

[3] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim, and J.-S. No, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022.

[4] D. Kim and C. Guyot, "Optimized privacy-preserving cnn inference with fully homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 2175–2187, 2023.

[5] D. Kim, J. Park, J. Kim, S. Kim, and J. H. Ahn, "Hyphen: A hybrid packing method and its optimizations for homomorphic encryption-based neural networks," *IEEE Access*, vol. 12, pp. 3024–3038, 2024.

[6] S. Meftah, B. H. M. Tan, C. F. Mun, K. M. M. Aung, B. Veeravalli, and V. Chandrasekhar, "Doren: Toward efficient deep convolutional neural networks with fully homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3740–3752, 2021.

[7] L. de Castro, R. Agrawal, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, C. Juvekar, and A. Joshi, "Does fully homomorphic encryption need compute acceleration?" *ArXiv CoRR*, vol. abs/2112.06396, 2021.

[8] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: Compiler and runtime for homomorphic evaluation of tensor programs," *ArXiv CoRR*, vol. abs/1810.00845, 2018.

[9] J.-W. Lee, E. Lee, Y.-S. Kim, and J.-S. No, "Rotation key reduction for client-server systems of deep neural network on fully homomorphic encryption," *Advances in Cryptology – ASIACRYPT 2023*, pp. 36–68, 2023.

[10] M. Li, S. S. M. Chow, S. Hu, Y. Yan, C. Shen, and Q. Wang, "Optimizing privacy-preserving outsourced convolutional neural network predictions," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1592–1604, 2022.

[11] L. Rovida and A. Leporati, "Encrypted image classification with low memory footprint using fully homomorphic encryption," *International journal of neural systems*, vol. 34, p. 2450025, 03 2024.

[12] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: an encrypted vector arithmetic language and compiler for efficient homomorphic computation," *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 546–561, 2020.

[13] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," *Advances in Cryptology – ASIACRYPT 2017*, pp. 409–437, 2017.

[14] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," *Advances in Cryptology – CRYPTO 2012*, pp. 868–886, 2012.

[15] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, vol. 2012/144, 2012.

[16] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, p. 309–325, 2012.

[17] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, pp. 34–91, 2019.

[18] S. Halevi and V. Shoup, "Design and implementation of HElib: a homomorphic encryption library," *Cryptology ePrint Archive*, 2020.

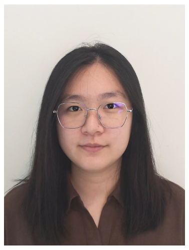
[19] "Lattigo v6," [Online]. Available: https://github.com/tuneinsight/lattigo, ePFL-LDS, Tune Insight SA, 2024.

[20] "Microsoft SEAL," [Online]. Available: https://github.com/Microsoft/SEAL, microsoft Research, Redmond, WA., 2023.

[21] T. Xie, H. Yamana, and T. Mori, "Che: Channel-wise homomorphic encryption for ciphertext inference in convolutional neural network," *IEEE Access*, vol. 10, pp. 107 446–107 458, 2022.

[22] J. H. Cheon, M. Kang, T. Kim, J. Jung, and Y. Yeo, "Batch inference on deep convolutional neural networks with fully homomorphic encryption using channel-by-channel convolutions," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–12, 2024.

VI. Biography Section



Jia-Lin Chan received a B.Sc. in software engineering from University Tunku Abdul Rahman, Malaysia, in 2022. She is currently pursuing her PhD in the Lee Kong Chian Faculty of Engineering and Science, UTAR, Malaysia. Her research interests include cryptography, GPU computing, deep learning, and data mining.

Wai-Kong Lee received a B.Eng. in electronics and an M.Eng.Sc. from Multimedia University, Malaysia in 2006 and 2009, respectively. He received a Ph.D. in engineering from Universiti Tunku Abdul Rahman, Malaysia in 2018. Prior to joining academia, he worked in several multi-national companies including Agilent Technologies (Malaysia) as an R&D engineer. His research interests include cryptography, numerical algorithms, GPU computing, the Internet of Things, and energy harvesting.

Denis-Chee-Keong Wong received B.Sc. and M.Sc. degree in Mathematics and the Ph.D. degree in Algebra/Coding Theory on 2001, 2004 and 2013, all from Universiti Sains Malaysia. He is now the assistant Professor and Head of Programme for the program, Master of Mathematics programme in UTAR. His research interests include Algebraic Coding Theory, Algebraic Combinatorics and cryptography.

Wun-She Yap is an associate professor in the Lee Kong Chian Faculty of Engineering and Science, UTAR, Malaysia. He received the Ph.D. degree from Multimedia University, Malaysia. He was the Chairperson of Centre for Cyber Security at UTAR from 2016 to 2020. He serves as the General Chair of ISPEC 2019 and has been invited to serve as program committees of a number of peerreviewed security conferences and the guest editors of special issues. His research interests include information security, cryptography and artificial intelligent.

Bok-Min Goi received his B.Eng. degree from University of Malaya (UM) in 1998, and the M.Eng.Sc. and Ph.D. degrees from Multimedia University (MMU), Malaysia in 2002 and 2006, respectively. He is now the Vice President and a senior professor in the Lee Kong Chian Faculty of Engineering and Science, Universiti Abdul Rahman Tunku (UTAR), Malaysia. He was the General Chair for ProvSec 2010 and CANS 2010, and the PC members for many crypto/security conferences. His research interests include cryptology, security protocols, information security, digital watermarking, computer networking and embedded systems design.