IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 10 March 2024, accepted 2 April 2024, date of publication 16 April 2024, date of current version 2 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3390053

RESEARCH ARTICLE

Innovative Homomorphic Sorting of Environmental Data in Area Monitoring Wireless Sensor Networks

NEETA B. MALVI AND N. SHYLASHREE, (Senior Member, IEEE)
Department of Electronics and Communication Engineering, RV College of Engineering (affiliated to Visvesvaraya Technological University, Belagavi), Bengaluru, Karnataka 560059, India

Corresponding author: Neeta B. Malvi (neetabm@rvce.edu.in)

ABSTRACT In many special cases, the data collected from wireless sensor networks are stored in encrypted form to provide the required privacy. Sorting is an essential operation on any stored data for orderly presentation and fast searching. In the case of cloud-stored data, sorting of the data can be delegated to the cloud server, employing suitable homomorphic encryption that supports sorting. This paper presents a new homomorphic sorting algorithm based on the Hardy-Littlewood-Polya rearrangement inequality. The associated homomorphic encryption scheme is accomplished using integer matrix keys generated based on Hermite Normal Form transformation. This work uses the homomorphic sort support encryption algorithm to securely sort the wireless sensor data stored in the cloud. Subsequently, the computation of the corresponding descriptive statistical values is securely outsourced to the Cloud Server. Our scheme saves the homomorphic sort execution time by about 30% compared to its nearest competing method.

INDEX TERMS Homomorphic encryption, homomorphic decryption, homomorphic sorting, parallel processing, privacy, wireless sensor network.

I. INTRODUCTION

Modern public cloud service platforms provide low-cost, high-volume data storage, authorized dissemination, and high-speed computational capabilities for their clients. However, information security is a genuine concern as data storage and processing are carried out in the public domain. Therefore, in general, sensitive data are stored in the cloud in encrypted form. Then, to utilize the computational capabilities of the cloud server, Homomorphic Encryption (HE) is adopted [1], [2], [3], which enables computations in the cipher domain that reflects the same results as and when carried out in the plain data domain. Thus, at present, many of the routine and heavy computations, like matrix operations [4], including machine learning, are securely outsourced to cloud servers [5], [6]. Sorting is an important operation on stored data for orderly presentation, for finding minimum/maximum, to reply to range queries, and as a prerequisite for the fast binary search. Homomorphic Sorting (HS) enables the sorting of encrypted data without the need for decryption. Thus HS provides a privacy-preserving and secure approach for sorting operations in scenarios where data confidentiality is required. Traditional sorting algorithms require direct access to the plaintext data to compare and rearrange the elements. However, in HS, the data remains encrypted throughout the sorting process. Thus the cloud client can delegate the sorting operations to the cloud without breaching security and privacy.

Homomorphic sorting algorithms typically utilize techniques such as Order Preserving Encryption (OPE) and homomorphic comparison (HC) networks [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23]. In OPE, if $x < y$, then $enc(x) < enc(y)$ for all x and y values in the data domain. Thus OPE facilitates

The associate editor coordinating the review of this manuscript and approving it for publication was Hosam El-Ocla.

HC. Subsequently, it can be used to implement homomorphic sorting. HE techniques ensure that the sorting process does not leak any information about the data being sorted to the cloud Server. In this paper, an entirely new technique based on Hardy-Littlewood-Polya (HLP) rearrangement inequality is used to determine the sort order in the cipher domain. The overall scheme is designated as 'Homomorphic Sort Support Encryption for Wireless Sensor Network' abbreviated as **HSSE-WSN**. The novelty of HSSE-WSN is its ability to encrypt floating-point plaintext data and then sort the encrypted floating-point data homomorphically in the cloud server. The paper is organized as follows. Section II gives the Review of the related work, and section III contains the working of HSSE-WSN. Section IV discusses the applications of HSSE-WSN. Comparative performance analysis is presented in section V, and sectionVI holds the conclusion.

II. RELATED WORK

Substantial work has been carried out on OPE that leads to HC and, subsequently, to HE. One of the early works to realize OPE is by Agrawal et al. [7], where the ciphertext is generated based on the piece-wise linear spline interpolation. Here, the data values are portioned into buckets of varying widths to match the corresponding data values. The main drawback of this method is the use of a complex data structure that acts as the encryption/decryption key. Additionally, the ciphertext data range is relatively large compared to that of the plaintext data. In [8], the authors have used the properties of Hyper Geometric Distribution (HGD) to achieve OPE. Here, the plaintext data is mapped into its equivalent ciphertext data based on the monotonically increasing function derived from HGD. The proposed HGD-based method provides a high level of security, but it is computationally expensive due to the use of recursive calls in its implementation. In [9], the encrypted value y is derived from the plaintext value x using the logarithmic expression $y = a*\log(x+b)$ with a and b as the secret key parameters. However, this elementary OPE scheme is vulnerable to the chosen plaintext and ciphertext attack unless the parameters are randomized frequently. In [10], the authors have used 'weighted random interval division' (WRID) to realize OPE for wearable systems. Here, the final weighted binary tree is constructed recursively by merging different temporary trees, constructed based on the data and the selected weights. The construction of multiple trees results in higher computational costs. In [11], HC (homomorphic comparison) is carried out by checking the sign of the result of subtraction in 2's complement form where MSB $= 1$ indicates that the sign is negative and MSB $=$ 0 means the sign of the result is positive. The entire operation is implemented using the bit-level homomorphic logic gates. Additionally, the authors have shown that, in their scheme, sorting methods based on comparison (like selection sort, bubble sort, etc.) and based on the divide and conquer rule (like quick sort, merge sort, etc.) have almost the same level of performance.

In [12], the authors have used 'Logistic Maps' for HC. Here, AES (or DES) is used for data privacy, and the logistic map is adopted to generate the matching codes for HC. Thus, it is a two-stage iterative process, so the run-time is relatively high. In [13], k-way sorting networks are adopted, instead of the conventional 2-way networks, to reduce the depth of comparison. These sorting networks are implemented using SIMD (Single Instruction Multiple Data) to speed up the operations. Since this method is based on CKKS homomorphic encryption scheme, round-off errors may occur due to floating point overflow. In [14], the authors have implemented a scalable homomorphic sorting scheme based on the 'polynomial rank sort algorithm.' Here, with batch mode operation, the time complexity is reduced from $O(n^2)$ to $O(n)$ for small values of n, where n is the number of elements to be sorted. However, this method requires a relatively large number of homomorphic multiplications, which reduces its efficiency for large values of n. In [15], bivariate polynomials are used for OPE, and then comparison circuits are built based on the encrypted polynomials representing the plaintext data. All operations are carried out in the finite field Fpd. Here, the evaluation of multiple bivariate polynomials increases the computational cost for large-sized plaintext data. In [16], the authors have presented the HC of floating point numbers using the continued fraction format where the precision of the floating point numbers can be accurately controlled. Here, the HC is built on top of the basic conventional HC of integers. However, the use of the continued fraction format requires iterative processing that results in higher computational cost.

In [17], the modified shell sort method and Fully Homomorphic Encryption (FHE) are combined to achieve homomorphic sorting. Here, the probability of sorting failure is made very small by increasing the runtime. Additionally, the main algorithm uses the bit level FHE in nested iterations, increasing the overall time complexity to a large extent. In [18], the authors have used 'Torus-based FHE (Fully Homomorphic Encryption)' to reduce the cost of bootstrapping as in conventional FHE schemes. Here, the bitwise comparison is achieved using the binary gates. Also, a modified bubble sort algorithm denoted by 'Addition-Sort' is implemented that can support homomorphic sorting of large arrays. However, the native TFHE scheme has no support for homomorphic multiplication, which has to be achieved indirectly using regular modular algebra over a finite field. This, incurs additional computational overhead.

In [19], the integers to be encrypted are converted into the 'Vector Field Elements' and these elements are then encrypted using the conventional FHE. Here, bivariate polynomial are used for order comparison. However, the length of the polynomials grows exponentially with respect to the data size, and to overcome this, the authors have used the block-wise approach, which increases the computational cost of each comparison. In [20], the authors have used word-wise encryption, unlike the bitwise conventional FHE. Here, iterative algorithms are introduced for comparison and min/max operations whereby top-k elements and the count of elements above a given threshold can be determined in the cipher domain. But the use of multiple iterations in the evaluation of polynomials increases the run time cost when the coefficients of the polynomials are large. In [21], a ring-based FHE scheme is used for HC of two numbers, namely, *a* and *b,* where a single ciphertext is generated that is used to get the comparison result in the cipher domain. The main disadvantage of this method is the individual plaintext elements are not encrypted for storage at the cloud server for possible access by authorized decrypters. In [22], the authors have presented an optimized 'compare-and-swap' algorithm based on FHE. Here, plaintext data is converted into a vector of plaintext slots using the Chinese Remainder Theorem, and the HC and swap operations are carried out in these slots. However, for the large-sized plaintext data, the corresponding plaintext slots and computations within these slots are also large and thus result in a multifold increase in the overall computational cost. In [23], HC for real numbers (both fixed point and floating point) is realized using the BGV (Brakerski-Gentry-Vaikuntanathan) scheme in the binary message space. The multiple binary circuits needed for comparison are executed according to SIMD to reduce the circuit depths and to increase the computational speed. However, the bit-level processing increases the computational cost when the operands are real numbers. In [24], word-level HC is achieved based on the bit-level FHE. Thus, a new technique called XCMP has been adopted to compare two integers in the cipher domain for equality as well as order comparison. Additionally, SIMD processing is implemented for faster computations. The authors have used polynomial representation for private comparison, which incurs a substantial increase in the computational cost. Overall, the main algorithm is highly iterative with increased time complexity.

Compared to the existing schemes, HSSE-WSN works at the word level (instead of bit-level), can handle floating point as well as integer data, uses the well-established library functions sort(...), find(...) and max(...) to speed up the operations. The novelty of HSSE-WSN is the use of Hardy-Littlewood-Polya (HLP) rearrangement inequality for homomorphic sorting.

III. PRELIMINARIES AND WORKING OF HSSE-WSN

A. ORGANIZATION OF WSN

The basic layout of the WSN is shown in Figure 1. The whole geographical area is divided into several zones, namely Zone 1, Zone 2, and so on, as shown in Figure 1(a). The Id of Zone u is taken as u itself for $u = 1, 2,$ and so on. Each zone is divided into n equal square-shaped sub-zones denoted by Area 1, Area 2,..., Area n, as shown in Figure 1(b). These areas are denoted by the indexed sequence of symbols, A(1), A(2),..., A(j),..., A(n) for easy representation. In HSSE-WSN, the range of n can be 4 to 10. In Figure 1(b), n = 9. In every sub-zone (Area j for $j = 1$ to n), a suitable sensor node is placed that senses the specific environmental data (as per the design) of that area, as shown in Figure 1(b).

A Cluster Head (CH) is placed, as usual, at the center of each zone, that is, in Area 5, as shown in Figure 1(a) and (b). The total number of zones depends on the total area and the range of the individual sensor nodes. In Figure 1(b), for each zone, we assume that node j fully covers Area j for j = 1 to n.

B. REPRESENTATION OF SENSOR DATA

The sensed data, from a zone, at a given time slot is represented by the data vector $\boldsymbol{D}$ as,

$$\boldsymbol{D} = [d\,(1)\,, d\,(2)\,, \ldots, d\,(j)\,, \ldots, d(n)] \tag{1}$$

Here, $d(j)$ is the sensed data corresponding to A(j) for $j = 1$ to n. Thus data of area A(j) is represented by $d(j)$. These $d(j)'s$ are collected, using TDMA, by the corresponding CH and arranged in the vector form as in (1). The CH, in turn, implements Homomorphic Sort Support Encryption (HSSE) of the plaintext data vector $\boldsymbol{D}$. The elements of $\mathbf{D}$ can be integers or floating point numbers. The encrypted data vector, denoted by $\boldsymbol{E}$, is transmitted from the CH to the designated CS for storage and further processing, as shown in Figure 2. In (1), $d(j)$ can be of any data type, like bytes, integers, doubles, etc.

C. SORT INDEX AND HOMOMORPHIC SORTING

When an array is sorted, its sort index gives the locations of the sorted elements from the unsorted array. In HSSE-WSN, Let $srt\,(\boldsymbol{D})$ be the sorted (in ascending order) version of the data vector $\boldsymbol{D}$ as,

$$srt\,(\boldsymbol{D}) = [s\,(1)\,, s\,(2)\,, \ldots, s\,(j)\,, \ldots, s(n)] \tag{2}$$

In HSSE-WSN, $srt\,(\boldsymbol{D})$ is obtained by the built-in sort(...) [25] function, which also gives the sort index vector of $\boldsymbol{D}$, as,

$$[srt\,(\boldsymbol{D}), \boldsymbol{siD}] = \text{sort}(\boldsymbol{D}) \tag{3}$$

The default direction of `sort(...)` is 'ascend'. In (3), $\boldsymbol{siD}$ is the *sort index vector* of D of length n, represented as,

$$\boldsymbol{siD} = [sid\,(1)\,, sid\,(2)\,, \ldots, sid\,(j)\,, \ldots, sid(n)] \tag{4}$$

$\boldsymbol{siD}$ is a permutation vector of integers in the range 1 to n. In (4), $sid\,(j)$ represents the jth element of $\boldsymbol{siD}$. The *sort index vector* $\boldsymbol{siD}$ is formed such that, for $j = 1$ to n,

$$s(j) = d\,(sid\,(j)) \tag{5}$$

In (5), $s(j)$ is the jth element of $srt\,(\boldsymbol{D})$, and $d\,(sid\,(j))$ is the $(sid\,(j))^{th}$ element of vector $\boldsymbol{D}$. In the vectorized index form [33], the above relation is written as,

$$srt\,(\boldsymbol{D}) = \boldsymbol{D}(\boldsymbol{siD}) \tag{6}$$

From (2) and (5), the minimum and the maximum data values are,

$$min\,(\boldsymbol{D}) = s\,(1) = d\,(sid\,(1)) \tag{7}$$

$$max\,(\boldsymbol{D}) = s\,(n) = d\,(sid\,(n)) \tag{8}$$

In Homomorphic Sorting (HS), the sorting operation is carried out in the cipher domain without the knowledge of

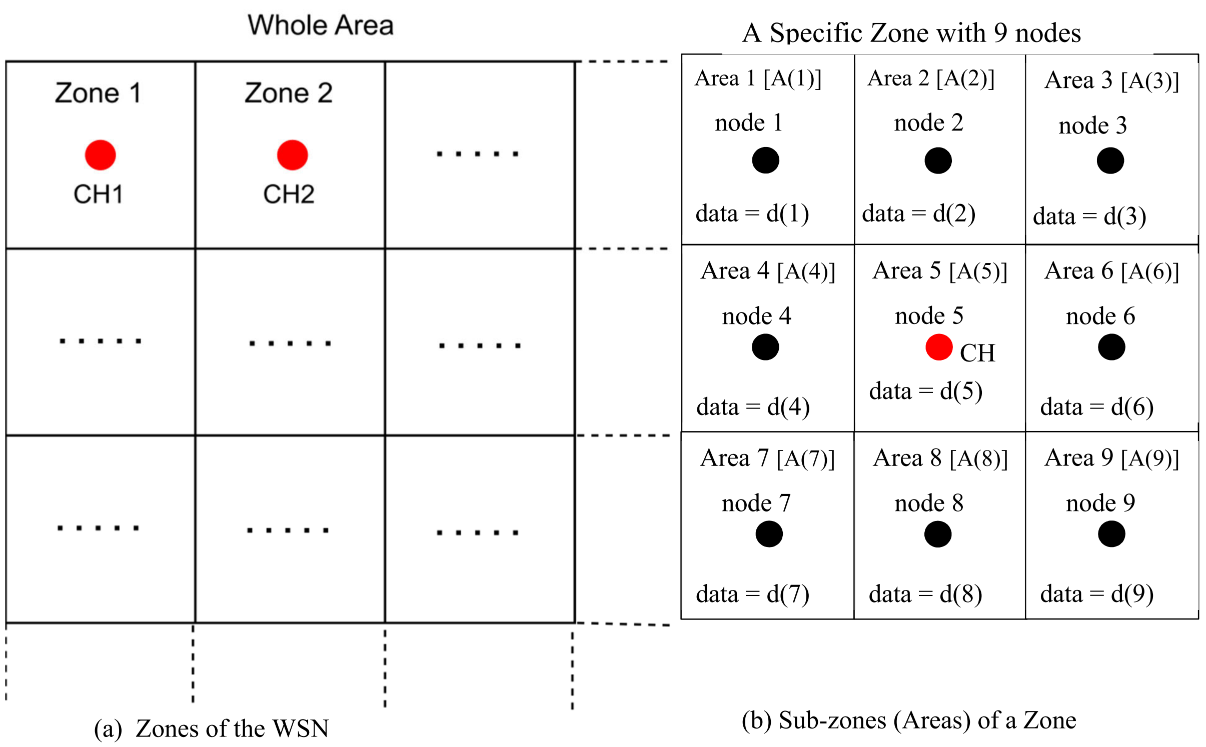
Whole Area
Zone 1
CH1
Zone 2
CH2
A Specific Zone with 9 nodes
Area 1 [A(1)] node 1 data = d(1)
Area 2 [A(2)] node 2 data = d(2)
Area 3 [A(3)] node 3 data = d(3)
Area 4 [A(4)] node 4 data = d(4)
Area 5 [A(5)] node 5 CH data = d(5)
Area 6 [A(6)] node 6 data = d(6)
Area 7 [A(7)] node 7 data = d(7)
Area 8 [A(8)] node 8 data = d(8)
Area 9 [A(9)] node 9 data = d(9)
(a) Zones of the WSN
(b) Sub-zones (Areas) of a Zone

FIGURE 1. **Organization of the WSN, zones and sub-zones.**

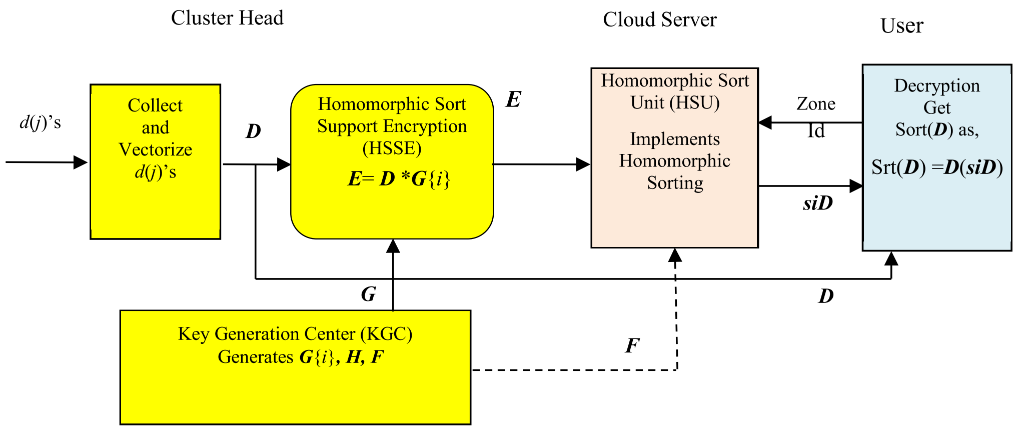
Cluster Head
Cloud Server
User
d(j)'s
Collect and Vectorize d(j)'s
D
Homomorphic Sort Support Encryption (HSSE) E= D *G{i}
E
Homomorphic Sort Unit (HSU) Implements Homomorphic Sorting
Zone Id
siD
Decryption Get Sort(D) as, Srt(D) =D(siD)
G
D
F
Key Generation Center (KGC) Generates G{i}, H, F

FIGURE 2. **Working arrangement of OPE-WSN.**

the original data or the decryption key. In HSSE-WSN, the Homomorphic Sorting Unit (HSU), housed in the CS, is designed to calculate the Sort Index vector ***siD*** of the original data vector ***D***. Once, ***siD*** is known to the user, who has access to ***D***, can determine *srt* (***D***) , based on (6). The HS operation is carried out for each zone of the WSN and the user can get the corresponding ***siD*** by quering CS with the Zone Id of that zone as shown in Figure 2.

D. HOMOMORPHIC SORT SUPPORT ENCRYPTION AND DECRYPTION KEYS

In HSSE-WSN, the Homomorphic Sort Support Encryption (HSSE) and decryption are carried out using integer matrices as keys. Integer matrix keys have smaller elements than scalar keys, and the computational overhead is relatively low. The cryptographic keys are generated by the Key Generation Center (KGC) at the CH.

1) GENERATION OF UNIMODULAR MATRIX

Initially, a unimodular matrix $\boldsymbol{U}$ of size $m \times m$ is generated using the Hermite Normal Form generation function `hermiteForm(...)` [26] as,

$$[\boldsymbol{U}, HNF] = \texttt{hermiteForm}(\boldsymbol{RIM}) \tag{9}$$

In (9), ***RIM*** is a Random Integer Matrix of size m×k. Matrix ***RIM*** is the secret seed chosen by the Key Generation Center, matrix $\boldsymbol{HNF}$ is the Hermite Normal Form of $\boldsymbol{RIM}$, and $\boldsymbol{U}$ is the unimodular matrix (det(U) $= \pm 1$) of size $m \times m$. In HSSE-WSN, the decryption matrix is designed to possess the Left Null Space (LNS) [27]. (The need for LNS will be explained in the next sub-section). Therefore, the parameter m is chosen greater than n (the length of the data vector D) as,

$$m = n + L \tag{10}$$

In (10), L is normally set to 2 so that $(m–n) = 2$. For higher security, a higher value of L can be selected at the cost of an increased ciphertext expansion ratio. In choosing $\boldsymbol{RIM}$, its column size parameter k is selected to be less than the row size parameter m so that the maximum element of $abs(\boldsymbol{U})$ is within a certain reasonable limit.

Now, the inverse of $\boldsymbol{U}$, denoted by $\boldsymbol{V}$, is calculated as $\boldsymbol{V} =$ inv($\boldsymbol{U}$). Since $\boldsymbol{U}$ is unimodular, the elements of $\boldsymbol{V}$ are integers, and we have,

$$\boldsymbol{UV} = \boldsymbol{I}_{\boldsymbol{m}\times\boldsymbol{m}} \tag{11}$$

Here, $\boldsymbol{I}_{m\times m}$ is the identity matrix of size $m \times m$.

2) GENERATION OF THE ENCRYPTION AND DECRYPTION KEYS

The encryption and decryption keys are obtained by partitioning matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ as,

$$\boldsymbol{U} = \begin{bmatrix} \boldsymbol{G}_{\boldsymbol{n}\times\boldsymbol{m}} \\ -- \\ \boldsymbol{W}_{(\boldsymbol{m}-\boldsymbol{n})\times\boldsymbol{m}} \end{bmatrix}, \ \boldsymbol{V} = \left[\boldsymbol{H}_{\boldsymbol{m}\times\boldsymbol{n}} | \boldsymbol{V2}_{\boldsymbol{m}\times(\boldsymbol{m}-\boldsymbol{n})}\right] \tag{12}$$

In (12), $\boldsymbol{H}_{\boldsymbol{m}\times\boldsymbol{n}}$ is the decryption key and $\boldsymbol{G}_{\boldsymbol{n}\times\boldsymbol{m}}$ is the basic encryption key.

The partitioned matrices will be substituted in (11). To match the partition as given by (12), the RHS of (11) is also partitioned as,

$$\boldsymbol{I}_{\boldsymbol{m}\times\boldsymbol{m}} = \begin{bmatrix} \boldsymbol{I}_{\boldsymbol{n}\times\boldsymbol{n}} & | & 0_{\boldsymbol{n}\times(\boldsymbol{m}-\boldsymbol{n})} \\ -- & | & -- \\ 0_{(\boldsymbol{m}-\boldsymbol{n})\times\boldsymbol{n}} & | & \boldsymbol{I}_{(\boldsymbol{m}-\boldsymbol{n})\times(\boldsymbol{m}-\boldsymbol{n}} \end{bmatrix} \tag{13}$$

On substituting (13) and (12) in (11), we get,

$$\begin{bmatrix} \boldsymbol{G}_{\boldsymbol{n}\times\boldsymbol{m}} \\ -- \\ \boldsymbol{W}_{(\boldsymbol{m}-\boldsymbol{n})\times\boldsymbol{m}} \end{bmatrix} \left[\boldsymbol{H}_{\boldsymbol{m}\times\boldsymbol{n}} | \boldsymbol{V2}_{\boldsymbol{m}\times(\boldsymbol{m}-\boldsymbol{n})}\right] = \begin{bmatrix} \boldsymbol{I}_{\boldsymbol{n}\times\boldsymbol{n}} & | & 0_{\boldsymbol{n}\times(\boldsymbol{m}-\boldsymbol{n})} \\ -- & | & -- \\ 0_{(\boldsymbol{m}-\boldsymbol{n})\times\boldsymbol{n}} & | & \boldsymbol{I}_{(\boldsymbol{m}-\boldsymbol{n})\times(\boldsymbol{m}-\boldsymbol{n}} \end{bmatrix} \tag{14}$$

From (14), we get,

$$\boldsymbol{G}_{\boldsymbol{n}\times\boldsymbol{m}}\boldsymbol{H}_{\boldsymbol{m}\times\boldsymbol{n}} = \boldsymbol{I}_{\boldsymbol{n}\times\boldsymbol{n}} \tag{15}$$

In (15), $\boldsymbol{G}_{\boldsymbol{n}\times\boldsymbol{m}}$ is the generalized inverse [28] of $\boldsymbol{H}_{\boldsymbol{m}\times\boldsymbol{n}}$.

$$\boldsymbol{W}_{(\boldsymbol{m}-\boldsymbol{n})\times\boldsymbol{m}}\boldsymbol{H}_{\boldsymbol{m}\times\boldsymbol{n}} = 0_{(\boldsymbol{m}-\boldsymbol{n})\times\boldsymbol{n}} \tag{16}$$

The property (16) means, $\boldsymbol{W}_{(\boldsymbol{m}-\boldsymbol{n})\times\boldsymbol{m}}$ is the left-null space of $\boldsymbol{H}_{\boldsymbol{m}\times\boldsymbol{n}}$.

3) RANDOMIZED CLONES OF G

In HSSE-WSN, $\boldsymbol{G}$ is the basic encryption key. However, to prevent Chosen Plaintext Attack (CPA) [29], the encryption key is randomized over successive encryption. To achieve randomization, both sides of Equation (16) are pre-multiplied by an arbitrary random integer matrix $\boldsymbol{R}\{1\}_{n\times(m-n)}$ to get,

$$\begin{aligned} \boldsymbol{R}\{1\}_{n\times(m-n)}\boldsymbol{W}_{(\boldsymbol{m}-\boldsymbol{n})\times\boldsymbol{m}}\boldsymbol{H}_{\boldsymbol{m}\times\boldsymbol{n}} &= \boldsymbol{R}\{1\}_{n\times(m-n)}0_{(m-n)\times n} \\ &= 0_{n\times n} \end{aligned} \tag{17}$$

After deleting the dimension subscripts, we get,

$$(\boldsymbol{R}\{1\}\boldsymbol{W})\,\boldsymbol{H} = 0 \tag{18}$$

The size of $(\boldsymbol{R}\{1\}\boldsymbol{W})$ is $n \times (m \times n) \times (m \times n) \times m = n \times m$. Now, Let matrix $\boldsymbol{G}\{1\}$ be formed as,

$$\boldsymbol{G}\{1\} = \boldsymbol{G} + \boldsymbol{R}\{1\}\boldsymbol{W} \tag{19}$$

Post multiplication of both sides of (19) by $\boldsymbol{H}$ **gives**,

$$\boldsymbol{G}\{1\}\boldsymbol{H} = \boldsymbol{GH} + \boldsymbol{R}\{1\}\boldsymbol{WH} \tag{20}$$

From (15), (18), and (20),

$$\boldsymbol{G}\{1\}\boldsymbol{H} = \boldsymbol{I}_{n\times n} \tag{21}$$

In (21), the size of $\boldsymbol{G}\{1\}$ is $(n \times m)$, which is equal to that of $\boldsymbol{G}$. In (19), $\boldsymbol{G}\{1\}$ is obtained by perturbing $\boldsymbol{G}$ by the random coefficient matrix $\boldsymbol{R}\{1\}$. Hence, $\boldsymbol{G}\{1\}$ is the randomized clone of $\boldsymbol{G}$. In our scheme HSSE-WSN, matrix $\boldsymbol{G}\{1\}$ is the first randomized encryption key.

In (19), $\boldsymbol{R}\{1\}$ is an arbitrary random matrix, and hence, the KGC can choose random matrices $\boldsymbol{R}\{2\}, \boldsymbol{R}\{3\}, \ldots, \boldsymbol{R}\{i\}, \ldots$, *etc*., which are dissimilar to $\boldsymbol{R}\{1\}$. Correspondingly, $\boldsymbol{G} + \boldsymbol{R}\{i\}\,\boldsymbol{W}$ can have dissimilar values as, $\boldsymbol{G}\{2\} = \boldsymbol{G} + \boldsymbol{R}\{2\}\boldsymbol{W}$, $\boldsymbol{G}\{\boldsymbol{3}\} = \boldsymbol{G} + \boldsymbol{R}\{3\}\boldsymbol{H}, \ldots$ and so on as,

$$\boldsymbol{G}\{i\} = \boldsymbol{G} + \boldsymbol{R}\{i\}\boldsymbol{W} \tag{22}$$

Similar to G{1} as in (21), it can be verified for $i = 1, 2, \ldots$, *etc*., that,

$$\boldsymbol{G}\{i\}\boldsymbol{H} = \boldsymbol{I}_{n\times n} \tag{23}$$

Here, the index value i, that identifies the randomized encryption key, is denoted as the randomization index. Equation (23) represents the fundamental relation between the cryptographic keys.

Example 1: Here is a simple numerical example that demonstrates the generation of matrix keys. The size parameters are $n = 4$, $m = 6$ and $k = 3$. The $\boldsymbol{RIM}$ used in (9) and the unimodular matrix $\boldsymbol{U}$, obtained from (9) as well as its inverse $\boldsymbol{V}$ are shown below.

$$\boldsymbol{RIM} = \begin{bmatrix} 3 & 3 & 2 \\ 3 & 1 & 2 \\ 2 & 2 & 2 \\ 3 & 1 & 1 \\ 2 & 1 & 2 \\ 1 & 2 & 2 \end{bmatrix} \quad \boldsymbol{U} = \left[\begin{array}{cccccc} 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 & 1 & -1 \\ 1 & 0 & 2 & -2 & 1 & -3 \\ \hdashline 0 & 1 & -1 & 0 & -1 & 1 \\ 0 & 0 & 5 & -2 & 0 & -4 \end{array}\right]$$

$$\boldsymbol{V} = \left[\begin{array}{cccc:cc} 3 & 3 & 2 & 1 & 0 & -2 \\ 3 & 1 & 2 & 0 & 1 & -1 \\ 2 & 2 & 2 & 0 & 0 & -1 \\ 3 & 1 & 1 & 0 & 0 & -1 \\ 2 & 1 & 2 & 0 & 0 & -1 \\ 1 & 2 & 2 & 0 & 0 & -1 \end{array}\right]$$

It can be verified that $\boldsymbol{UV} = \boldsymbol{VU} = \boldsymbol{I}_{6\times 6}$.Now, matrices $\boldsymbol{U}$ and V are partitioned to get $\boldsymbol{G}$, $\boldsymbol{W}$, and $\boldsymbol{H}$ as specified by (12) as,

$$\boldsymbol{G} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 & 1 & -1 \\ 1 & 0 & 2 & -2 & 1 & -3 \end{bmatrix} \quad \boldsymbol{W} = \begin{bmatrix} 0 & 1 & -1 & 0 & -1 & 1 \\ 0 & 0 & 5 & -2 & 0 & -4 \end{bmatrix}$$

$$\boldsymbol{H} = \begin{bmatrix} 3 & 3 & 2 & 1 \\ 3 & 1 & 2 & 0 \\ 2 & 2 & 2 & 0 \\ 3 & 1 & 1 & 0 \\ 2 & 1 & 2 & 0 \\ 1 & 2 & 2 & 0 \end{bmatrix}$$

Now, it can be verified that $\boldsymbol{GH} = \boldsymbol{I}_{4\times 4}$.To get randomized encryption keys G{1} and G{2}, we generate two random integer matrices R{1} and R{2}, each with size 4×2, and then use Equation (22). Matrices R{1}, R{2}, G{1}, and G{2} are shown below.

$$\boldsymbol{R}\{1\} = \begin{bmatrix} 4 & 5 \\ 2 & 4 \\ 4 & 4 \\ 3 & 2 \end{bmatrix} \quad \boldsymbol{R}\{2\} = \begin{bmatrix} 3 & 5 \\ 6 & 4 \\ 6 & 6 \\ 5 & 3 \end{bmatrix}$$

$$\boldsymbol{G}\{1\} = \begin{bmatrix} 0 & 4 & 22 & -10 & -4 & -17 \\ 0 & 2 & 19 & -8 & -3 & -14 \\ 0 & 4 & 17 & -9 & -3 & -13 \\ 1 & 3 & 9 & -6 & -2 & -8 \end{bmatrix}$$

$$\boldsymbol{G}\{2\} = \begin{bmatrix} 0 & 3 & 23 & -10 & -3 & -18 \\ 0 & 6 & 15 & -8 & -7 & -10 \\ 0 & 6 & 25 & -13 & -5 & -19 \\ 1 & 5 & 12 & -8 & -4 & -10 \end{bmatrix}$$

Here, it can be verified that $\boldsymbol{G}\{1\}\boldsymbol{H} = \boldsymbol{G}\{2\}\boldsymbol{H} = \boldsymbol{I}_{4\times 4}$.

E. HOMOMORPHIC ENCRYPTION

The homomorphic encryption of the data vector $\boldsymbol{D}$ is carried out by post-multiplying it by $\boldsymbol{G}\{\mathrm{i}\}$ to get the cipher vector $\boldsymbol{E}$ as,

$$\boldsymbol{E} = \boldsymbol{DG}\{i\} \tag{24}$$

In (24), the size of $\boldsymbol{E}$ is $(1\times \mathrm{n})\times(n \times m) = 1\times m$. The encryption process is similar to the Hill-Cipher, but the CPA attack inherent in the Hill-Cipher is eliminated by choosing different i's (randomization indices) for successive encryptions. The encrypted vector $\boldsymbol{E}$ is transmitted and stored in the cloud server for further processing. Since the encryption process is carried out at each CH, with $(n \times m)$ multiplications per encryption, the CHs should have higher energy resources.

1) CIPHERTEXT EXPANSION RATIO

Ciphertext expansion ratio (CER) is the ratio of the ciphertext size to plaintext size. Higher the CER, higher would be the associated communication and computational cost. In HSSE-WSN, the size of plaintext matrix **D** is (1×n), and that of the ciphertext matrix **E** is $(1\times m)$. Taking m = n + L, the CER is (m/n) = (n+L)/n is very nearly equal to one as n is relatively large compared to L. The CER that is nearly one reduces the communication overhead while transmitting the ciphertext from the CH to the CS.

2) SECURITY OF HOMOMORPHIC ENCRYPTION

Brute Force Attack: The sizes of Sender's keys $\boldsymbol{G}\{\mathrm{i}\}$'s and Receiver's key $\boldsymbol{H}$ are $n \times m$ and $m \times n$, respectively. Hence, there are $m\times n$ elements in a key. Assuming the average size of an element in a key as two digits (decimal), the probability of guessing a single element is 10^{-2}. Therefore, the probability of correctly guessing all the elements (PCGall) is PCGall = $10^{-2(m\times n)}$, which is very low for single digit m and n values. For example, with $m = 6$ and $n = 4$, PCGall = 10^{-48}. Thus, the brute force attack is almost eliminated.

In HSSE-WSN, for the given data length n, the value of m can be increased by choosing a higher L as given by (10). Since PCGall = $10^{-2(m\times n)}$, it an increase in m decreases PCGall exponentially. Then the resistance to the brute force attack increases at the cost of increased CER, which is given by (m/n). Then the computational overhead also increases

3) RESISTANCE TO THE CHOSEN CIPHERTEXT ATTACK (CCA)

From the encryption process (24), by knowing the ciphertext $\boldsymbol{E}$ and the plaintext $\boldsymbol{D}$,it is not possible to find G{i} as (24) is an under-determined equation in terms of $\boldsymbol{E}$ and $\boldsymbol{D}$. Even when G{i} is obtained somehow, it is not used in the next encryption. Thus even the adaptive CCA fails.

F. HOMOMORPHIC DECRYPTION

The homomorphic decryption of $\boldsymbol{E}$ is accomplished by post-multiplying it by $\boldsymbol{H}$ to get the plaintext data vector $\boldsymbol{B}$ as,

$$\boldsymbol{B} = \boldsymbol{EH} \tag{25}$$

In (25), the size of B is $(1\times \mathrm{m})\times(m \times n) = 1\times n$. The resulting $\boldsymbol{B}$ is exactly same as $\boldsymbol{D}$. the Correctness of the decryption can be verified by substituting for $\boldsymbol{E}$ in (25) from (24) and then applying the fundamental relation (23) as follows.

$$\boldsymbol{B} = \boldsymbol{EH} = \boldsymbol{DG}\{\boldsymbol{i}\}\boldsymbol{H} = \boldsymbol{DI}_{n\times n} = \boldsymbol{D}$$

Thus, the decryption is carried out as,

$$\boldsymbol{D} = \boldsymbol{EH} \tag{26}$$

The end user who has the decryption key $\boldsymbol{H}$ in his possession can decrypt $\boldsymbol{E}$ to get the original plaintext data $\boldsymbol{D}$ in the case of simple decryption.

G. HSSE-WSN WORKING

1) BASIC PRINCIPLE

The working of HSSE-WSN is based on the Hardy-Littlewood-Polya (HLP) rearrangement inequality [30]. Consider the data vector $\boldsymbol{D}$ of length is n, as in (1), and a column vector $\boldsymbol{Y}$ also of length n. Then of According to the HLP inequality,

$$\begin{aligned} Sort\ (\boldsymbol{D}, \text{'}ascend\text{'}) * sort\ (\boldsymbol{Y}, \text{'}ascend\text{'}) &\geq \boldsymbol{DY} \\ sort\ (\boldsymbol{D}, \text{'}ascend\text{'}) * sort\ (\boldsymbol{Y}, \text{'}descend\text{'}) & \end{aligned} \tag{27}$$

In (27), each product term is the scalar (inner) product of the corresponding vectors. Thus, from the HLP inequality, the maximum of $\boldsymbol{DY}$ over different possible rearrangements is,

$$max\ (\boldsymbol{DY}) = Sort\ (\boldsymbol{D}, \text{'}ascend\text{'}) * sort\ \left(\boldsymbol{Y}, \text{'}ascend\text{''}\right)$$

On dropping the '*ascend*' qualifier, we have,

$$max\ (\boldsymbol{DY}) = srt\ (\boldsymbol{D}) * srt\ (\boldsymbol{Y}) \tag{28}$$

In (28), $srt\ (\boldsymbol{D})$ and$srt\ (\boldsymbol{Y})$ are the sorted versions of the row vector $\boldsymbol{D}$ and the column vector $\boldsymbol{Y}$.

For a given $\boldsymbol{D}$, let $\boldsymbol{Y}max$ be the specific rearrangement (permutation) of $\boldsymbol{Y}$ that maximizes the scalar product $\boldsymbol{DY}$. Then,

$$\boldsymbol{DY}max = max\ (\boldsymbol{DY}) \tag{29}$$

From (29) and (28),

$$\boldsymbol{DY}max = srt\ (\boldsymbol{D}) * srt\ (\boldsymbol{Y}) \tag{30}$$

From (6), we have $srt\ (\boldsymbol{D})$ equal to $\boldsymbol{D}(\boldsymbol{siD})$.Similarly, $srt\ (\boldsymbol{Y})$ is equal to $\boldsymbol{Y}max(\boldsymbol{siY}max)$. Therefore, Equation (30) can be rewritten as,

$$\boldsymbol{D} * \boldsymbol{Y}max = \boldsymbol{D}(\boldsymbol{siD}) * \boldsymbol{Y}max(\boldsymbol{siY}max) \tag{31}$$

The scalar product $\boldsymbol{D}*\boldsymbol{Y}max$ is invariant when the same permutation is applied to both $\boldsymbol{D}$ and $\boldsymbol{Y}max$. Therefore, when the pair $\boldsymbol{D}$ and $\boldsymbol{Y}max$ are permuted by the index vector by $\boldsymbol{siD}$ which is the sort index of $\boldsymbol{D}$, we have,

$$\boldsymbol{D}*\boldsymbol{Y}max = \boldsymbol{D}(\boldsymbol{siD})*\boldsymbol{Y}\text{max}(\boldsymbol{siD}) \tag{32}$$

From (31) and (32), we have,

$$\boldsymbol{D}\ (\boldsymbol{siD}) * \boldsymbol{Y}max\ (\boldsymbol{siY}max) = \boldsymbol{D}(\boldsymbol{siD})*\boldsymbol{Y}\text{max}(\boldsymbol{siD}) \tag{33}$$

Comparison of the LHS and the RHS of (33) shows that when the scalar product is maximum, the sort index of $\boldsymbol{Y}max$ is equal to the sort index of $\boldsymbol{D}$. In (33), the sort index $\boldsymbol{siD}$ is a row vector whereas $\boldsymbol{siY}max$ is a column vector. However, the individual elements and their order are same for both $\boldsymbol{siD}$ and $\boldsymbol{siY}max$. Therefore, while equating the sort indices of $\boldsymbol{D}$ and $\boldsymbol{Y}max$, the column vector $\boldsymbol{siY}max$ is transposed to make its dimension compatible with $\boldsymbol{siD}$**.** Thus the equality of index vectors is expressed as,

$$\boldsymbol{siD} = (\boldsymbol{siY}max)^{\boldsymbol{T}} \tag{34}$$

In (34), the transpose operation is used to take care of the row-column mismatch. Equation (34) means, for a given data vector $\boldsymbol{D}$**,** its sort index vector $\boldsymbol{siD}$ can be determined inter alia by finding the sort index vector of $\boldsymbol{Y}max$ where $\boldsymbol{Y}max$ is that permutation of $\boldsymbol{Y}$,which maximizes the scalar (dot) product $\boldsymbol{D} * \boldsymbol{Y}$.From (34) and (6),

$$srt\ (\boldsymbol{D}) = \boldsymbol{D}\ (\boldsymbol{siD}) = \boldsymbol{D}(\boldsymbol{siY}max) \tag{35}$$

It should be noted that when $\boldsymbol{siY}max$ is used as the sort index vector, $\boldsymbol{D}\ (\boldsymbol{siY}max) = \boldsymbol{D}\left(\boldsymbol{siY}max^{\boldsymbol{T}}\right)$**.**

a) USER INTERACTION

Once $\boldsymbol{siD}$ is known, $srt\ (\boldsymbol{D})$ can be obtained by the End user according to (6). The End user should have access to $\boldsymbol{D}$ unles the user is the CH itself. The End user queries the CS by sending the Zone Id of the WSN and gets back the matching sID from the CS, from which the corresponding $srt(\boldsymbol{D})$ is obtained.

In general, the equality represented by (28) can be stated as a corollary of HLP inequality as follows.

Corollary 1:

When the scalar product $\boldsymbol{DY}$ reaches its maximum over all possible permutation iterations over $\boldsymbol{Y}$, the sort vectors $\boldsymbol{siD}$ and $\boldsymbol{siY}max$ are equal in the sense $srt\ (\boldsymbol{D}) = \boldsymbol{D}(\boldsymbol{siY}max)$.

2) HOMOMORPHIC SORT UNIT

The data vector $\boldsymbol{D}$ is encrypted as $\boldsymbol{E}$ (the cipher vector) and then stored in the CS. The Homomorphic Sort Unit (HSU) is housed within the CS. The HSU generates $\boldsymbol{siD}$ with vector $\boldsymbol{E}$ as the input. Another critical input used by the HSU is the cipher matrix denoted by the symbol $\boldsymbol{F}$, which is the full collection of Encoded Permutation Vectors. Matrix $\boldsymbol{F}$ is generated by the KGC and stored in the CS, as indicated in Figure 2, and it is readily available to the HSU.

a: FORMATION OF MATRIX F

Initially, the KGC chooses a Random Seed Vector $\boldsymbol{RSV}$ of distinct integer elements in the range specified by the Lower Bound (LB) and the Upper Bound (UB). The size of $\boldsymbol{RSV}$ is set to $n \times 1$. The 'distinct elements' property assures unique sort index vector $srt\ (\boldsymbol{RSV})$ for a given $\boldsymbol{RSV}$. Then, all possible permutations of $\boldsymbol{RSV}$ are generated in the column format using the function perms($\boldsymbol{RSV}$) [31] as,

$$\boldsymbol{P} = [perms(\boldsymbol{RSV})]^T \tag{36}$$

The size of the *perms* matrix $\boldsymbol{P}$ is n$\times$ (n!). Matrix $\boldsymbol{P}$ is represented in terms of its columns as,

$$\boldsymbol{P} = [p\ (1)\ , p(2), \ldots, p(k), \ldots, p(n!)] \tag{37}$$

In (37), $p(k)$is the k^{th} column of $\boldsymbol{P}$ for $k = 1$ to $n!$. The size of $p(k)$ is $n \times 1$, and it represents the k^{th} permutation of the seed vector $\boldsymbol{RSV}$. Now, cipher matrix $\boldsymbol{F}$ is formulated as,

$$\boldsymbol{F} = \boldsymbol{HP} \tag{38}$$

In (38), $\boldsymbol{H}$ is the decryption matrix of size $(m \times n)$, and the size of $\boldsymbol{F}$ is $(m \times n) \times (n \times n!) = (m \times n!)$. The cipher matrix $\boldsymbol{F}$, generated by the KGC, is stored in the CS. This is a one-time operation at the beginning of the HSSE-WSN session. In (38), both $\boldsymbol{H}$ and $\boldsymbol{P}$ are private to the KGC. Therefore $\boldsymbol{F}$, which is the product of $\boldsymbol{H}$ and $\boldsymbol{P}$, reveals no information about the individual factors $\boldsymbol{H}$ and $\boldsymbol{P}$. Thus the matrix $\boldsymbol{F}$ at CS does not cause any security breach.

b: DETERMINATION OF ***siD*** *AT HSU*

The HSU in CS, with matrices $\boldsymbol{E}$ and $\boldsymbol{F}$ as inputs, generates the product matrix $\boldsymbol{Q}$ as, in (39),

$$\boldsymbol{Q} = \boldsymbol{EF} \tag{39}$$

In (39), matrix $\boldsymbol{E}$ is the encryption of the data matrix $\boldsymbol{D}$, and the size of $\boldsymbol{E}$ is $(1\times m)$. The size of $\boldsymbol{F}$ is $(m\times n!)$. Hence the size of $\boldsymbol{Q}$ is $(1\times n!)$. Similar to as explained in the case of $\boldsymbol{F}$, the knowledge of Q does not leak any information about $\boldsymbol{D}$, $\boldsymbol{G}\{i\}$ or $\boldsymbol{H}$. Now, substituting for $\boldsymbol{E}$ from (24) and $\boldsymbol{F}$ from (38), we have,

$$\boldsymbol{Q} = \boldsymbol{D} * \boldsymbol{G}\{i\}\boldsymbol{HP} \tag{40}$$

Then, substituting for $\boldsymbol{G}\{i\}\boldsymbol{H}$ from (23) in (40), we get,

$$\boldsymbol{Q} = \boldsymbol{DI}_{n\times n}\boldsymbol{P} = \boldsymbol{DP} \tag{41}$$

On expanding $\boldsymbol{P}$ from (37), we have,

$$\begin{aligned}\boldsymbol{Q} &= \boldsymbol{D}\left[p(1), p(2), \ldots, p(k), \ldots, p(n!)\right]\\ &= \left[\boldsymbol{D}p(1), \boldsymbol{D}p(2), \ldots, \boldsymbol{D}p(k), \ldots, \boldsymbol{D}p(n!)\right]\end{aligned} \tag{42}$$

In (42), $\boldsymbol{Q}$ is a row vector of size $(1\times n!)$. On expressing $\boldsymbol{Q}$ in terms of its individual elements $q(1)$, $q(2)$, etc, Equation (42) can be rewritten as,

$$\boldsymbol{Q} = \left[q(1), q(2), \ldots, q(k), \ldots, q(n!)\right] \tag{43}$$

Thus, for $k = 1$ to $n!$, the k^{th} element of $\boldsymbol{Q}$ is given by,

$$q(k) = \boldsymbol{D}p(k) \tag{44}$$

In (44), the size of $\boldsymbol{D}$ is $(1\times n)$ and that of $p(k)$ is $(n\times 1)$. Thus $q(k)$ is the scalar product of $\boldsymbol{D}$ and $p(k)$. In (44), the scalar product $q(k)$ is obtained corresponding to all possible permutations of $p(k)$ when k varies over the range 1 to $n!$.

Let q_k attain its maximum when the index $k = kmax$. Then, $kmax$ can be obtained using the max(...) [32] function as,

$$\left[q_{kmax}, kmax\right] = max(\boldsymbol{Q}) \tag{45}$$

Once $kmax$ is known, the corresponding $P(kmax)$ is obtained as the $kmax^{\text{th}}$ column of matrix $\boldsymbol{P}$. From the column vector $P(kmax)$, its sort index vector $\boldsymbol{si}Pkmax$ is obtained using the sort(...) function as,

$$\left[srt\left(P(kmax)\right), \boldsymbol{si}Pkmax\right] = sort\left(P(kmax)\right) \tag{46}$$

Then, from Corollary 1 of section III-A, we have,

$$\boldsymbol{siD} = (\boldsymbol{si}Pkmax)^{\boldsymbol{T}} \tag{47}$$

Once $\boldsymbol{siD}$ is generated by the HSU, the user receives it from the CS, and he can get $srt(\boldsymbol{D})$ as,

$$srt(\boldsymbol{D}) = \boldsymbol{D}(\boldsymbol{siD}) = \boldsymbol{D}(\boldsymbol{si}Pkmax) \tag{48}$$

Here, we get $\boldsymbol{siD}$ from $\boldsymbol{si}Pkmax$ which in turn is obtained after determining $kmax$ using (45) where the matrix $\boldsymbol{Q} = \boldsymbol{EF}$, is in the cipher domain. Thus, homomorphic sorting is achieved in HSU.

3) ALGORITHM-HSU

The sequence of operations involved in determining $\boldsymbol{ssiD}$ is given in **Algorithm-HSU** as follows.

Algorithm-HSU

Inputs: Cipher vector E of size $(1\times m)$ and the cipher matrix **F** of size $(m\times n!)$.

Output: Sort index vector $\boldsymbol{si}Pkmax$ of size $(n\times 1)$, $\boldsymbol{siD}$ and $srt(\boldsymbol{D})$.

```
1. Compute the product $Q = EF$.
2. Find $kmax$ using the max(...) function as
   $[q_{kmax}, kmax] = max(\mathbf{Q})$.
3. Get $P(kmax)$, the $kmax^{th}$ column of matrix $P$.
4. Get $siPkmax$ using (46).
5. Get $siD$ as $siD = (siPkmax)^T$.
6. Get $srt(D)$ as, $srt(D) = D(siD)$.
7. Over.
```

Example 2: This example calculates $srt(\boldsymbol{D})$ using the encryption/decryption keys of Example 1. Here, n = 4, m = 6, and the data vector $\boldsymbol{D}$ is taken as $\boldsymbol{D} = [23, 15, 25, 12]$. The permutation seed vector RSV is chosen as RSV = $[2, 4, 1, 6]^{\text{T}}$. The perms matrix $\boldsymbol{P}$, obtained using (30) is,

$$\boldsymbol{P} = \begin{bmatrix} 6&6&6&6&6&6&1&1&1&1&1&1&4&4&4&4&4&4&2&2&2&2&2&2\\ 1&1&4&4&2&2&6&6&4&4&2&2&6&6&1&1&2&2&6&6&1&1&4&4\\ 4&2&1&2&1&4&4&2&6&2&6&4&1&2&6&2&6&1&1&4&6&4&6&1\\ 2&4&2&1&4&1&2&4&2&6&4&6&2&1&2&6&1&6&4&1&4&6&1&6 \end{bmatrix}$$

The cipher matrix $\boldsymbol{F}$, obtained as $\boldsymbol{F} = \boldsymbol{HP}$ is found to be, as shown in the equation at the bottom of the next page.

The encrypted vector $\boldsymbol{E}$, calculated using $\boldsymbol{E} = \boldsymbol{DG}1\}$ is found to be $\boldsymbol{E} = [12, 258, 1324, -647, -236, -1022]$.

Now, $\boldsymbol{Q} = \boldsymbol{EF}$ is found to be,

$\boldsymbol{Q} = [277, 251, 247, 260, 241, 280, 237, 211, 257, 205, 251, 225, 231, 244, 281, 229, 284, 219, 209, 248, 259, 233, 268, 203]$

Now, $[q_{kmax}, kmax] = max(\boldsymbol{Q})$ calculation gives, $[284, 17] = \max(\boldsymbol{Q})$. That is, $kmax = 17$. Therefore, we have,

$P(kmax) = P(17) = 17^{th}$ column of P which is $[4, 2, 6, 1]^{\text{T}}$. Now, from (46), $\boldsymbol{si}P(kmax) = [4, 2, 1, 3]^{\text{T}}$ which means, $\boldsymbol{siD} = [4, 2, 1, 3]$ and $srt(\boldsymbol{D}) = \boldsymbol{D}([4, 2, 1, 3]) =$ [12], [15], [23], [25].

4) SELECTION OF THE PERMUTATION SEED VECTOR RSV

If the elements of RSV are large, then the elements of its permutation vector $P'_k s$ are also large, which in turn can result in very large values of the elements of $\boldsymbol{H}$ [see Eq. (32)]. This, in turn, may cause integer overflow or computational slow-down. This disadvantage is avoided by keeping the elements of RSV within certain Lower and Upper Bounds (LB and

UB). Additionally, when n (the length of $\boldsymbol{RSV}$) is large, vector $\boldsymbol{RSV}$ is chosen to have a mix of negative as well as positive integers so that the scalar product terms q_k's are not too big.

5) COMPUTATIONAL COMPLEXITY OF HSU

Computations of the cryptographic keys, G{i}, H, and F are one-time operations, and hence their complexities are excluded from the discussion. The main run time complexity of HSU operations is due to the matrix product calculation $\boldsymbol{Q} = \boldsymbol{EF}$ and finding *kmax* via *max*($\boldsymbol{Q}$). The length of $\boldsymbol{Q}$ is $n!$, and the time complexity is O($n!$). The size of $\boldsymbol{E}$ is (1xm), and that of $\boldsymbol{F}$ is ($m \times n!$). Hence, the number of integer multiplications is O($m \times n!$)≈ O($n \times n!$). Thus, for substantially large values of n, the HSU operation requires high-speed processors with GPU accelerators and parallel processing. Additionally, when n is large, the 2D array $\boldsymbol{F}$ may not fit in the system RAM. Then, $\boldsymbol{F}$ can be stored on the hard drive [34] and can be processed block-wise.

IV. APPLICATIONS OF HSSE-WSN

A. MAPPING OF MINIMUM AND MAXIMUM DATA AREAS

Once the sort index vector $\boldsymbol{siD}$ is known, the minimum of the data vector $\boldsymbol{D}$ is obtained using (7) as,

$$min\,(\boldsymbol{D}) = s\,(1) = d\,(\mathrm{sid}\,(1))$$

Since the sensed data of area A (j) is represented by $d(j)$, the minimum data area is $\mathrm{A}\,(\mathrm{sid}\,(1))$. Similarly, the maximum data area is $\mathrm{A}\,(\mathrm{sid}\,(n))$. In the case of repeating entries in $\boldsymbol{D}$, the *max*($\boldsymbol{Q}$) will have multiple maximum values. However Equation (6) holds true where siD is calculate using (46). These calculations can be extended to all the zones of the WSN to get the maximum and minimum values of every zone.

B. RANGE QUERY

In this case, the objective is to find the distribution of $d(j)$'s (data values) between two pre-defined levels, namely the LL (Lower Level) and the UL (Upper Level). To achieve this objective, we introduce the augmented data sequence, $\boldsymbol{C}$**,** which is formed by appending LL and UL to the normal data vector D as,

$$\begin{aligned}\boldsymbol{C} &= [\boldsymbol{D}, LL, UL] \\ &= [d\,(1)\,, d\,(2)\,, \ldots, d\,(j)\,, \ldots, d\,(n)\,, LL, UL] \qquad (49)\end{aligned}$$

Here, the size of $\boldsymbol{C}$ is $1\times(n{+}2)$. In (49), $UL > LL$. On expanding $\boldsymbol{C}$ in terms of its elements c(1), c(2), etc., in (49), we get,

$$\begin{aligned}\boldsymbol{C} &= [c\,(1)\,, c\,(2)\,, \ldots, c\,(j)\,, \ldots, c\,(n)\,, c(n+1), c(n+2)] \\ &= [d\,(1)\,, d\,(2)\,, \ldots, d\,(j)\,, \ldots, d\,(n)\,, LL, UL] \qquad (50)\end{aligned}$$

In (50), c(n+2) > c(n+1). From (50), we see that,

$$c(j) = \begin{cases} d\,(j) \ \text{ for } 1 \le j \le n \\ LL \text{ for } j = n+1 \\ UL \text{ for } j = n+2 \end{cases} \qquad (51)$$

That is,

$$\left.\begin{aligned} c\,(n+1) &= LL \\ c\,(n+2) &= UL \end{aligned}\right\} \qquad (52)$$

Now, the augmented cryptographic keys, $\boldsymbol{AG}\{i\}$ (augmented $G\{i\}$) of size $(n+2)$x$(n+4)$, $\boldsymbol{AH}$ (augmented H) of size $(n+4)$x$(n+2)$, and $\boldsymbol{AF}$ (augmented F) of size $(n+2)$x$(n+2)!$, are regenerated to match the size of the augmented data vector $\boldsymbol{C}$ instead of $\boldsymbol{D}$. The augmented vector $\boldsymbol{C}$ is encrypted as $\boldsymbol{AE}$ (augmented $\boldsymbol{E}$) and stored in the CS. Now the HSU operates on $\boldsymbol{AE}$ and gives out the sort index vector $\boldsymbol{siC}$ of $\boldsymbol{C}$. The end user who receives $\boldsymbol{siC}$ can determine the ranges of the individual data elements and the corresponding Areas covered by them as follows.

Let $\boldsymbol{siC}$ be represented as,

$$\begin{aligned}\boldsymbol{siC} = [&sic\,(1)\,, sic\,(2)\,, \ldots, sic\,(j)\,, \ldots, sic\,(n)\,, \\ &sic\,(n+1)\,, sic(n+2)] \qquad (53)\end{aligned}$$

Vector $\boldsymbol{siC}$ is a permutation (shuffled version) of the natural number sequence $[1, 2, \ldots, n, n+1, n+2]$. Let the sorted vector $srt\,(\boldsymbol{C})$ be represented as,

$$\begin{aligned}srt\,(\boldsymbol{C}) = [&srtc\,(1)\,, srtc(2)]\,, \ldots, srtc\,(j)\,, \\ &\ldots, srtc\,(n+1)\,, srtc(n+2)] \qquad (54)\end{aligned}$$

The basic relation between $srt\,(\boldsymbol{C})$ and $\boldsymbol{siC}$ is, $srt\,(\boldsymbol{C}) = \boldsymbol{C}\,(\boldsymbol{siC})$. On applying this relation to their individual elements as given by (54) and (53), we have, for $j = 1$ to $(n+2)$,

$$srtc\,(j) = c\,(sic\,(j)) \qquad (55)$$

The elements $srtc\,(j)'\,s$ are in the non-decreasing order. Now consider a specific index JU in the range 1 to $(n{+}2)$, then for those indices less than JU, we have,

$$srtc\,(j) \le srtc\,(JU) \ \text{ for } j < JU \qquad (56)$$

From (56) and (55),

$$c\,(sic\,(j)) \le c\,(sic\,(JU)) \ \text{ for } j < JU \qquad (57)$$

$$\boldsymbol{F} = \begin{bmatrix} 31 & 29 & 34 & 35 & 30 & 33 & 31 & 29 & 29 & 25 & 25 & 23 & 34 & 35 & 29 & 25 & 31 & 26 & 30 & 33 & 25 & 23 & 31 & 26 \\ 27 & 23 & 24 & 26 & 22 & 28 & 17 & 13 & 19 & 11 & 17 & 13 & 20 & 22 & 25 & 17 & 26 & 16 & 14 & 20 & 19 & 15 & 22 & 12 \\ 22 & 18 & 22 & 24 & 18 & 24 & 22 & 18 & 22 & 14 & 18 & 14 & 22 & 24 & 22 & 14 & 24 & 14 & 18 & 24 & 18 & 14 & 24 & 14 \\ 23 & 21 & 23 & 24 & 21 & 24 & 13 & 11 & 13 & 9 & 11 & 9 & 19 & 20 & 19 & 15 & 20 & 15 & 13 & 16 & 13 & 11 & 16 & 11 \\ 21 & 17 & 18 & 20 & 16 & 22 & 16 & 12 & 18 & 10 & 16 & 12 & 16 & 18 & 21 & 13 & 22 & 12 & 12 & 18 & 17 & 13 & 20 & 10 \\ 16 & 12 & 16 & 18 & 12 & 18 & 21 & 17 & 21 & 13 & 17 & 13 & 18 & 20 & 18 & 10 & 20 & 10 & 16 & 22 & 16 & 12 & 22 & 12 \end{bmatrix}$$

Now, find the index JU such that,

$$sic\left(JU\right)=\left(n+2\right) \tag{58}$$

JU satisfying (58) is accomplished using the `find(...)` [35] function as,

$$JU=find\left(\boldsymbol{siC}==\left(n+2\right)\right) \tag{59}$$

Substituting for $sic\left(JU\right)$ in (57) from (58), we have,

$$c\left(sic\left(j\right)\right)\leq c\left(n+2\right)\ \text{ for } j<JU \tag{60}$$

From (60) and (52),

$$c\left(sic\left(j\right)\right)\leq UL \text{ for } j<JU \tag{61}$$

Similarly, it can be shown that,

$$c\left(sic\left(j\right)\right)\geq UL \text{ for } j>JU \tag{62}$$

From (58), with $j = JU$, we have $sic\left(j\right) = \left(n+2\right)$. Therefore,

$$c\left(sic\left(j\right)\right)=c\left(n+2\right)\ \text{ for } j=JU \tag{63}$$

From (63) and (52),

$$c\left(sic\left(j\right)\right)=c\left(n+2\right)=UL \text{ for } j=JU \tag{64}$$

Equations (61), (62), and (64) can be jointly stated as,

$$\left.\begin{array}{c} c\left(sic\left(j\right)\right)\leq UL \text{ for } j<JU \\ c\left(sic\left(j\right)\right)=c\left(n+2\right)=UL \text{ for } j=JU \\ c\left(sic\left(j\right)\right)\geq UL \text{ for } j>JU \end{array}\right\} \tag{65}$$

Now, let us find the index JL such that,

$$sic\left(JL\right)=\left(n+1\right) \tag{66}$$

Similar to (59), JL is obtained as,

$$JL=find\left(\boldsymbol{siC}==\left(n+1\right)\right) \tag{67}$$

From (67) and (52),

$$c\left(sic\left(JL\right)\right)=c\left(n+1\right)=LL \tag{68}$$

Then, similar to (65), it can be shown that,

$$\left.\begin{array}{c} c\left(sic\left(j\right)\right)\leq LL \text{ for } j<JL \\ c\left(sic\left(j\right)\right)=LL \text{ for } j=JL \\ c\left(sic\left(j\right)\right)\geq LL \text{ for } j>JL \end{array}\right\} \tag{69}$$

Now, in (65) and (69), the elements of $\boldsymbol{C}$ can be replaced by the elements of $\boldsymbol{D}$ based on (51) and (52). From (64) and (68), $c\left(sic\left(j\right)\right) = c\left(n+2\right) = UL$ for $j = JU$, and $c\left(sic\left(j\right)\right) = c\left(n+1\right) = LL$ for $j = JL$. Here, $j = JL$ and $j = JU$ cover the elements $c\left(n+1\right)$ and$c\left(n+2\right)$ of vector $\boldsymbol{C}$. Therefore, $j{\neq}JL$ and $j{\neq}JU$ should cover other elements of $\boldsymbol{C}$. From (51), we know that the other elements of $\boldsymbol{C}$ are same as those of D in the sequential order. Hence, in (65) and (69), $c\left(sic\left(j\right)\right)$ can be replaced by $d\left(sic\left(j\right)\right)$ when $j{\neq}JL$ and $j{\neq}JU$. Thus, (65) and (69) can be rewritten in terms of the elements C as,

$$\left.\begin{array}{c} d\left(sic\left(j\right)\right)\leq UL \text{ for } j<JU \\ d\left(sic\left(j\right)\right)\geq UL \text{ for } j>JU \end{array}\right\} \tag{70}$$

$$\left.\begin{array}{c} d\left(sic\left(j\right)\right)\leq LL \text{ for } j<JL \\ d\left(sic\left(j\right)\right)\geq LL \text{ for } j>JL \end{array}\right\} \tag{71}$$

Equations (70) and (71) can be combined as,

$$\left.\begin{array}{c} d\left(sic\left(j\right)\right)\leq LL \text{ for } j<JL \\ LL\leq d\left(sic\left(j\right)\right)\leq UL \text{ for } JL<j<JU \\ d\left(sic\left(j\right)\right)\geq UL \text{ for } j>JU \end{array}\right\} \tag{72}$$

The relations given by (72) are expressed picturesquely in Figure 3. Here, the range of j is divided into three sections as $\left(j<JL\right), \left(JL<j<JU\right)$ and $\left(j>JU\right)$. The corresponding $d\left(sic\left(j\right)\right)$ ranges are shown in the respective sections.

Based on (72), the Areas covered by data vector $\boldsymbol{D}$ can be classified into three groups as,

$$\left.\begin{array}{c} \text{data values } \leq LL\text{correspond to } A\left(sic\left(j\right)\right)'\text{s for } j<JL \\ LL\leq \text{data values} \leq UL \text{ correspond to } A\left(sic\left(j\right)\right) \\ '\text{s for } JL<j<JU \\ \text{data values } \geq UL \text{ correspond to } A\left(sic\left(j\right)\right)'\text{s for } j>JU \end{array}\right\} \tag{73}$$

Thus, by knowing the $\boldsymbol{siC}$ vector, the Areas covered by $\boldsymbol{D}$ can be mapped according to the data ranges specified by LL and UL.

The Two level range query explained in this section can be summarized by the following algorithm.

Algorithm Range-Query

Inputs: Lower and Upper threshold levels LL and UL.
Sort index vector $\boldsymbol{siC}$ of size $1\text{x}(n+2)$ corresponding to the augmented vector $\boldsymbol{C}$ as specified by (50).
Output: Partitioned data elements according to the three ranges as given by (72).
Corresponding classification of Areas as given by (73).

```
1. Get index $JL$ as, $JL = find\left(\boldsymbol{siC} == \left(n+1\right)\right)$.
2. Get index $JU$ as, $JU = find\left(\boldsymbol{siC} == \left(n+2\right)\right)$.
3. Get the elements of $\boldsymbol{D}$ in the three ranges using (72).
4. Get the corresponding Areas in the three ranges
   using (73).
5. Over.
```

1) COMPUTATIONAL COMPLEXITY OF RANGE QUERY

The size vector C is (n+2). Hence, similar to as explained in III.G.5, computationa complexity of the Range Query is (n+2)×(n+2)!

Example 3: Here, $n = 9$ and the data vector is taken as,

$$D=\left[42\ 54\ 58\ 16\ 56\ 49\ 34\ 4\ 28\right] \tag{74}$$

The threshold levels LL and UL are set at $LL = 17$ and $UL = 40$. Hence the augmented vector $\boldsymbol{C}$ is,

$$C=\left[D, LL, UL\right]=\left[42\ 54\ 58\ 16\ 56\ 49\ 34\ 4\ 28\ 17\ 40\right] \tag{75}$$

The sort vector $\boldsymbol{siC}$ is found to be,

$$siC{=}\left[8\ 4\ 10\ 9\ 7\ 11\ 1\ 6\ 2\ 5\ 3\right] \tag{76}$$

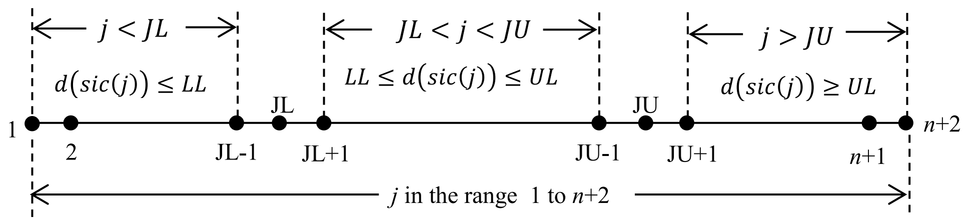

FIGURE 3. Division of the ***siC*** index space ***j*** into 3 non-overlapping sections.

TABLE 1. Execution time in milliseconds.

$n \rightarrow$	4	5	6	7	8	9	10
HSSE-WSN	0.4159	0.5451	0.6983	2.0635	19.7135	331.1243	9712.5437
Huang's Method	0.6239	0.8177	1.0475	3.0953	29.5703	232.5214	5733.8342
Lee's Method	1.1256	1.3239	2.0632	7.6984	42.9984	419.2266	5504.6076
Wang's Method	4.5300	6.2200	8.1500	17.2223	78.9973	850.2605	7415.3400

In this example, $(n + 1) = 10$ and $(n + 2) = 11$. Hence, $JL = find\,(\boldsymbol{siC} == (n+1)) = 3$ and $JU = find\,(\boldsymbol{siC} == (n+2)) = 6$. That is, the index positions of 10 and 11 in $\boldsymbol{siC}$ are 3 and 6. Thus, $sic(3) = 10$ and $sic(6) = 11$. Now the index space j (in the range 1 to 11) is split as,

$$\left.\begin{array}{r} j's = [1, 2] \text{ corresponding to } j < 3 \\ j's = [4, 5] \text{ corresponding to } 3 < j < 6 \\ j's = [7, 8, 9, 10, 11] \text{ corresponding to } j > 6 \end{array}\right\} \quad (77)$$

From (76) and (77), the corresponding sic(j)'s are,

$$\left.\begin{array}{r} sic\,(j)\,'s = [8, 4] \text{ corresponding to, } j < 3 \\ sic\,(j)\,'s = [9, 7] \text{ corresponding to, } 3 < j < 6 \\ sic\,(j)\,'s = [1, 6, 2, 5, 3] \text{ corresponding to } j > 6 \end{array}\right\} \quad (78)$$

Then, based on (72), with $LL = 17$ and $UL = 45$,

$$\left.\begin{array}{r} [d\,(8)\,, d(4)] \leq 17 \\ 17 \leq [d\,(9)\,, d\,(7)] \leq 45 \\ [d\,(1)\,, d\,(6)\,, d\,(2)\,, d\,(5)\,, d(3)] \geq 45 \end{array}\right\} \quad (79)$$

It can be verified that (79) is indeed true by comparing the values of (77) with the actual values of $\boldsymbol{D}$ as in (74).

C. DIGITAL SIGNATURE

Let the liberally estimated minimum and maximum among $d(j)$'s be denoted by *dLmin* and *dLmax*. In the Range query, the parameters *LL* and *UL* represent the Lower and Upper Levels selected by the CH (the client). Now, the digital signature scheme for HSSE-WSN is formulated by choosing *LL* substantially less than *dLmin* and choosing *UL* substantially greater than *dLmax* as,

$$LL = dLmin - -Loffset \quad (80)$$

$$UL = dLmax + Uoffset \quad (81)$$

In (80), *Loffset* is an appropriate positive numerical offset on the lower side. Similarly, in (81), *Uoffset* is one on the higher side. Then, vector $\boldsymbol{C}$ is formed by the CH as in (49) and is sent to the CS. Let the End User (EU) be the CH for minimum communication load. After the sorting process in the CS, the sort index vector $\boldsymbol{siC}$ is received by the CH (which is now the EU). From $\boldsymbol{siC}$, $\boldsymbol{srt}(\boldsymbol{C})$ is obtained as $\boldsymbol{srt}(\boldsymbol{C}) = \boldsymbol{C}(\boldsymbol{siC})$. Then, the signature verification is carried out by checking that $\min(\boldsymbol{C}) = LL$ and $\max(\boldsymbol{C}) = UL$. If these two conditions are not satisfied, it means a security breach involving confidentiality, integrity, or some other error. *LL* and *UL* are chosen once per session and remain constant for successive encryptions.

Additionally, by choosing *LL* and *UL* as non zeros, the formation of $\boldsymbol{C}$ with two non-zero elements generates non-zero ciphertext $\boldsymbol{E}$, even when the data vector $\boldsymbol{D}$ is all zeros. Otherwise, according to (24), $\boldsymbol{E}$ would be all zeros, thereby exposing all zero $\boldsymbol{D}$ to the CS.

V. COMPARATIVE PERFORMANCE

Here, the homomorphic sort execution times of HSSE-WSN, Huang's method [24], Lee's method [17],and Wang's method [18] are compared. The data type elements to be sorted are taken as 64-bit floating point numbers. In the comparison experiment, the size n of the plaintext data vector is varied from $n = 4$ to 10 with step size $=1$. The corresponding execution times are determined, and the corresponding values are shown in Table 1 and the plots are shown in Figure 4.

Execution times are machine-dependent, and the packages used for calculations. Therefore, the numerical values shown in Table 1 are relative. From Table 1, it can be seen that HSSE-WSN outperforms the other three methods for n less than or equal to 8. For $n > 8$, the size of $n!$ is relatively large, and the execution time increases accordingly. The graphical representation of the execution times is shown in Figure 4.

The percentage saving in the execution times for $n = 8$ is calculated as,

Percentage saving in the execution time Huang's method $=$ 100$*$(29.5703-19.7135)/29.5703 $= 33.33\%$.

Percentage saving in the execution time Lee's method $=$ 100$*$(42.9984-19.7135)/42.9984 $= 54.15\%$.

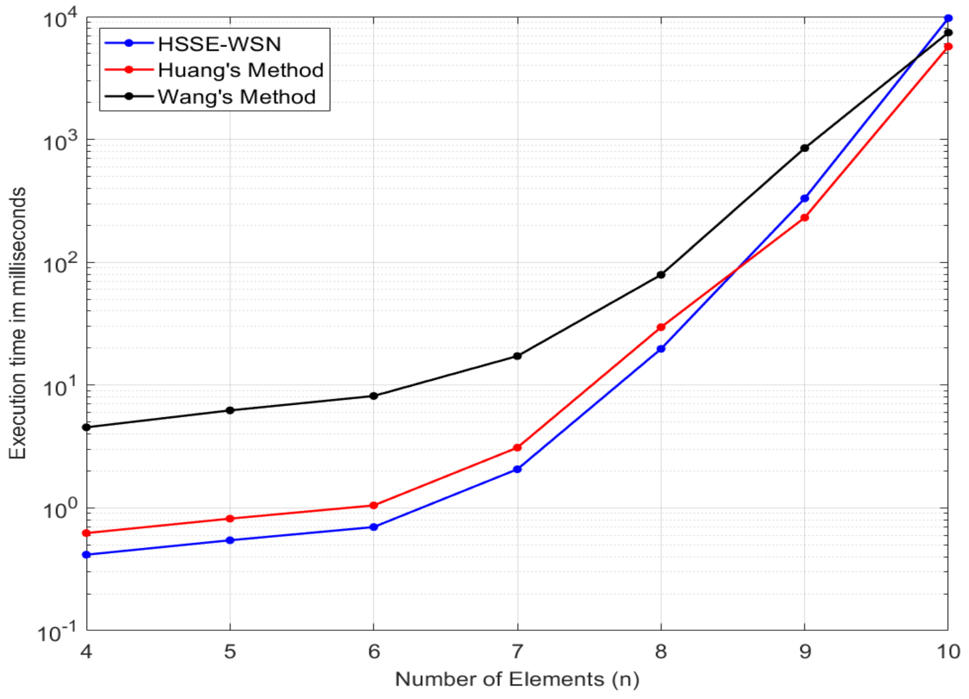

FIGURE 4. Execution times in homomorphic sorting.

Percentage saving in the execution time Wang's method = 100∗(78.9973-19.7135)/ 78.9973= 75.05%.

The reasons for higher time complexities of the three methods Huang's [24], Lee's [17], and Wang's [18], at lower values of n (n≤8) are because these methods use bit-level FHE, whereas HSSE-WSN uses the matrix-based method for FHE. Additionally, the algorithms of the other three methods use multiple iterations, which are responsible for their increased computational overhead.

VI. CONCLUSION

An entirely new method of homomorphic sorting based on the Hardy-Littlewood-Polya rearrangement inequality is presented. The sorting scheme can handle integer as well as double data types efficiently. The percentage saving in the homomorphic sort execution time is about 30% compared to its nearest competitor. Here, the encryption process is randomized to prevent the chosen ciphertext attack. In our homomorphic encryption scheme, the ciphertext expansion ratio is nearly equal to one, and it is independent of the data size. Additionally, the innovative homomorphic range detection scheme presented here is efficient and provides a convenient means of range-based mapping of the corresponding zones of the associated wireless sensor network. Even though the time complexity of our method is exponential, the powerful processors at the cloud servers, where the homomorphic sorting would be implemented, can handle the computational load with reasonable latency. Our homomorphic sorting scheme is easily amenable to parallel processing, and hence the sorting speed can be substantially improved using GPU-assisted computations.

In future work, preprocessing operations to remove the noise, sensitivity analysis, latency aspect, and parallel processing will be taken care of. The proposed scheme will demonstrate outstanding performance when these limitations are eliminated.

REFERENCES

[1] T. V. T. Doan, M.-L. Messai, G. Gavin, and J. Darmont, "A survey on implementations of homomorphic encryption schemes," *J. Supercomput.*, vol. 79, no. 13, pp. 15098–15139, Sep. 2023, doi: 10.1007/s11227-023-05233-z.

[2] K. Munjal and R. Bhatia, "A systematic review of homomorphic encryption and its contributions in healthcare industry," *Complex Intell. Syst.*, vol. 9, no. 4, pp. 3759–3786, Aug. 2023, doi: 10.1007/s40747-022-00756-z.

[3] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. P. Fitzek, and N. Aaraj, "Survey on fully homomorphic encryption, theory, and applications," *Proc. IEEE*, vol. 110, no. 10, pp. 1572–1609, Oct. 2022, doi: 10.1109/JPROC.2022.3205665.

[4] S. Wang and H. Huang, "Secure outsourced computation of multiple matrix multiplication based on fully homomorphic encryption," *KSII Trans. Internet Inf. Syst.*, vol. 13, no. 11, pp. 5616–5630, 2019.

[5] M. Nocker, D. Drexel, M. Rader, A. Montuoro, and P. Schottle, "HE-MAN–Homomorphically encrypted machine learning with oNnx models," 2023, *arXiv:2302.08260*.

[6] S. S. Gowri, S. Sadasivam, N. H. Priya, and T. A. D. Priyan, "Secured machine learning using approximate homomorphic scheme for healthcare," in *Proc. Int. Conf. Intell. Syst. Commun., IoT Secur. (ICIS-CoIS)*, Coimbatore, India, Feb. 2023, pp. 361–364, doi: 10.1109/ICIS-CoIS56541.2023.10100547.

[7] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order-preserving encryption for numeric data," in *Proc. SIGMOD Conf.*, G. Weikum, A. C. K'onig, and S. Debloch, Eds. New York, NY, USA: ACM, 2004, pp. 563–574, doi: 10.1145/1007568.1007632.

[8] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, ''Order-preserving symmetric encryption,'' in *Advances in Cryptology—EUROCRYPT* (Lecture Notes in Computer Science), vol. 5479, A. Joux, Ed. Cologne, Germany: Springer, Apr. 2009, pp. 224–241, doi: 10.1007/978-3-642-01001-9_13.

[9] E. Khoury, M. Medlej, C. A. Jaoude, and C. Guyeux, ''Novel order preserving encryption scheme for wireless sensor networks,'' in *Proc. IEEE Middle East North Afr. Commun. Conf. (MENACOMM)*, Jounieh, Lebanon, Apr. 2018, pp. 1–6, doi: 10.1109/MENACOMM.2018.8371028.

[10] R. Gui, L. Yang, and G. Xiaolin, ''An order-preserving encryption scheme based on weighted random interval division for ciphertext comparison in wearable systems,'' *Sensors*, vol. 22, no. 20, p. 7950, 2022, doi: 10.3390/s22207950.

[11] A. Chatterjee and I. Sengupta, ''Searching and sorting of fully homomorphic encrypted data on cloud,'' *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 981, Oct. 2015. [Online]. Available: https://eprint.iacr.org/2015/981

[12] H. Dai, H. Ren, Z. Chen, G. Yang, and X. Yi, ''Privacy-preserving sorting algorithms based on logistic map for clouds,'' *Secur. Commun. Netw.*, vol. 2018, pp. 1–10, Sep. 2018, doi: 10.1155/2018/2373545.

[13] S. Hong, S. Kim, J. Choi, Y. Lee, and J. H. Cheon, ''Efficient sorting of homomorphic encrypted data with k-way sorting network,'' *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 4389–4404, 2021, doi: 10.1109/TIFS.2021.3106167.

[14] G. S. Çetin, E. Savas, and B. Sunar, ''Homomorphic sorting with better scalability,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 760–771, Apr. 2021, doi: 10.1109/TPDS.2020.3030748.

[15] I. Iliashenko and V. Zucca, ''Faster homomorphic comparison operations for BGV and BFV,'' in *Proc. Privacy Enhancing Technol.*, 2021, pp. 246–264, doi: 10.2478/popets-2021-0046.

[16] H. Chung, M. Kim, A. A. Badawi, K. M. M. Aung, and B. Veeravalli, ''Homomorphic comparison for point numbers with user-controllable precision and its applications,'' *Symmetry*, vol. 12, no. 5, p. 788, May 2020, doi: 10.3390/sym12050788.

[17] J.-W. Lee, Y.-S. Kim, and J.-S. No, ''Analysis of modified shell sort for fully homomorphic encryption,'' *IEEE Access*, vol. 9, pp. 126198–126215, 2021, doi: 10.1109/ACCESS.2021.3110868.

[18] C. Wang, J. Chen, X. Zhang, and H. Cheng, ''An efficient fully homomorphic encryption sorting algorithm using addition over TFHE,'' in *Proc. IEEE 28th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Nanjing, China, Jan. 2023, pp. 226–233, doi: 10.1109/ICPADS56603.2022.00037.

[19] B. H. M. Tan, H. T. Lee, H. Wang, S. Ren, and K. M. M. Aung, ''Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields,'' *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 6, pp. 2861–2874, Nov. 2021, doi: 10.1109/TDSC.2020.2967740.

[20] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee, ''Numerical method for comparison on homomorphically encrypted numbers,'' in *Proc. ASIACRYPT*, 2019, pp. 415–445, doi: 10.1007/978-3-030-34621-8_15.

[21] W.-J. Lu, J.-J. Zhou, and J. Sakuma, ''Non-interactive and output expressive private comparison from homomorphic encryption,'' in *Proc. Asia Conf. Comput. Commun. Secur.*, May 2018, pp. 67–74, doi: 10.1145/3196494.3196503.

[22] C.-C. Huang, J.-N. Ji, and M.-D. Shieh, ''On compare-and-swap optimization for fully homomorphic encrypted data,'' in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2021, pp. 1–4, doi: 10.1109/ISCAS51556.2021.9401078.

[23] J. Basilakis and B. Javadi, ''Efficient parallel binary operations on homomorphic encrypted real numbers,'' *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 1, pp. 507–519, Jan. 2021, doi: 10.1109/TETC.2019.2906047.

[24] H. Huang, Y. Wang, L. Wang, H. Ge, and Q. Gu, ''Secure word-level sorting based on fully homomorphic encryption,'' *J. Inf. Secur. Appl.*, vol. 71, Dec. 2022, Art. no. 103372, doi: 10.1016/j.jisa.2022.103372.

[25] Sort. *Sort Array Elements*. Accessed: Jun. 6, 2023. [Online]. Available: https://in.mathworks.com/help/MATLAB/ref/sort.html

[26] Accessed: Jun. 6, 2023. [Online]. Available: https://in.mathworks.com/help/symbolic/hermiteform.html

[27] *Lecture 10: The Four Fundamental Subspaces*. Accessed: Feb. 7, 2024. [Online]. Available: https://ocw.mit.edu/courses/18-06sc-linear-algebra-fall-2011/62a9db9eeab190694d40afe4734068ca_MIT18_06SCF11_Ses1.10sum.pdf

[28] *LEC 6G Inverse*. Accessed: Feb. 7, 2024. [Online]. Available: https://www.sjsu.edu/faculty/guangliang.chen/Math253S20/lec6ginverse.pdf

[29] J. Katz and Y. Lindell, *Introduction to Modern Cryptography: Principles and Protocols*. Boca Raton, FL, USA: CRC Press, 2007.

[30] G. H. Hardy J. E. Littlewood, and G. PPólyalya, *Inequalities* (Cambridge Mathematical Library), 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 1952, ch. 368, sec. 10.2.

[31] Perms. *All Possible Permutations*. Accessed: Jun. 6, 2023. [Online]. Available: https://in.mathworks.com/help/MATLAB/ref/perms.html

[32] Max. *Maximum Elements of Array*. Accessed: Jun. 6, 2023. [Online]. Available: https://in.mathworks.com/help/MATLAB/ref/max.html

[33] Matrix Indexing in MATLAB. *By Steve Eddins and Loren Shure, MathWorks*. Accessed: Jun. 6, 2023. [Online]. Available: https://in.mathworks.com/company/newsletters/articles/matrix-indexing-in-MATLAB.html

[34] *Processing Large NumPy Arrays With Memory Mapping*. Accessed: Jun. 6, 2023. [Online]. Available: https://ipython-books.github.io/48-processing-large-numpy-arrays-with-memory-mapping/

[35] Accessed: Jun. 6, 2023. [Online]. Available: https://in.mathworks.com/help/matlab/ref/find.html

NEETA B. MALVI received the B.E. and M.Tech. degrees in digital communication and networking and telecommunication engineering from Visvesvaraya Technological University, Karnataka, India.

She is currently a Research Scholar with Visvesvaraya Technological University, Belagav. Since 2014, she has been an Assistant Professor with the Electronics and Communication Engineering Department, RV College of Engineering, Bengaluru, Karnataka. Her research interests include network security, wireless sensor networks, and machine learning.

N. SHYLASHREE (Senior Member, IEEE) is currently an Associate Professor with the Department of Electronics and Communication Engineering, RV College of Engineering, Bengaluru. She is having 18 years of teaching experience. She has research publications in 40 international journals (out of which 12 journals are SCI journals), six Springer book chapters, and ten international conferences. She got the grant for two Indian patents and one USA patent in the area of cryptography. She has also got the grant for two Indian patents in the area of VLSI. She is also the coauthor of the *Network Theory*, *Engineering Statistics and Linear Algebra*, and *Control Engineering*. She has funded projects on chalcogenide materials and many consultancy projects and has delivered many technical talks on VLSI. She has delivered lectures as a Subject Matter Expert in VTU e-Shikshana and EDUSAT Program. Her research interests include network analysis, analysis and design of digital circuits, digital VLSI design, analog and mixed mode VLSI design, low-power VLSI design, cryptography and network security, statistics and linear algebra, and control engineering. She is a Life Member of ISTE and IETE, a fellow of ISVE, and a Senior Member of the IEEE CAS Secretary-Bengaluru Section. She was a recipient of the Best Ph.D. Thesis Award for the year 2016–2017 in electronics and communication engineering from BITES. She received the Best IEEE Researcher Award from the IEEE-AGM Meeting held in 2021 from Bengaluru IEEE Section. She also received the Best Researcher Award from ISTE-RVCE Chapter, in 2022. She has received the Best Paper Award from IEEE-ICERECT held in 2015 at Mandya. She received Best faculty advisor award from IEEE CASS Bangalore section in the year 2023. She was a recipient of the International Travel Grant under the SERB Young Research Scholar Category.