International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 9, Number 1, pages 111–129, January 2019

Semi-Order Preserving Encryption Technique for Numeric Database

Saleh Ahmed
Graduate School of Engineering, Hiroshima University
1-7-1 Kagamiyama, Higashi-Hiroshima, Japan
Email: d162694@hiroshima-u.ac.jp

Annisa
Department of Computer Science, Bogor Agricultural University
Jln. Meranti, Kampus IPB Darmaga, Bogor 1668, Indonesia
Email: annisa@myfamily.web.id

Asif Zaman
Department of Computer Science and Engineering, University of Rajshahi
Rajshahi-6205, Bangladesh
Email: asifzaman3180@yahoo.com

Zhan Zhang
Graduate School of Engineering, Hiroshima University
1-7-1 Kagamiyama, Higashi-Hiroshima, Japan
Email: m154277@hiroshima-u.ac.jp

Kazi Md. Rokibul Alam
Department of Computer Science and Engineering
Khulna University of Engineering and Technology, Khulna, Bangladesh
Email: rokib@cse.kuet.ac.bd

and

Yasuhiko Morimoto
Graduate School of Engineering, Hiroshima University
1-7-1 Kagamiyama, Higashi-Hiroshima, Japan
Email: morimoto@mis.hiroshima-u.ac.jp

Received: February 1, 2018
Revised: May 5, 2018
Revised: July 31, 2018
Revised: September 14, 2018
Accepted: October 4, 2018
Communicated by Hiroyuki Sato

Abstract

Order preserving encryption techniques are treated as some of the most efficient encryption schemes for securing numeric data in a database. Such schemes are popular because they resolve performance degradation issues, which are significant problems in database encryption.

However, in some applications, the order itself is sensitive information, and should be hidden. Conventional order preserving encryption techniques published so far, did not consider this issue. Therefore, in this study, we consider three techniques that protect the order information and also show good performance. The three methods hide the data order such that comparison operators can be handled efficiently and performance degradation can be prevented. Our methods work on the top of an order preserving encryption scheme and enhance the security of data. Experimental results demonstrate the efficiency and effectiveness of the proposed three methods.

Keywords: Order preserving encryption scheme (OPES), Semi-order, Privacy preserving, Numeric database.

1 Introduction

With the increasing demand of big-data processing and cloud computing, various organizations and database engineers currently emphasize on the security and privacy of sensitive data such as employee salary, age, and expenditure. Such sensitive data are often stored in a database. Several encryption techniques have been proposed for preserving the privacy of sensitive data [12, 4]. However, such techniques degrade the performance of query execution performed on the encrypted database; significant amount of time is spent for decrypting each tuple before computing conditional operations provided as query a parameter. To address this problem, different variants of the order preserving encryption scheme (OPES) and some of its variants have been proposed in few articles [1, 2, 3, 4, 10, 6, 9].

We can also improve the performance of database queries by preserving the index information of the original numeric values. However, in some database applications, the ordered index of numerical data itself is considered as sensitive information. For example, some institutions may consider the merit position of an individual as sensitive information that should not be disclosed to public. In this regard, we considered semi-order preserving techniques, which perturb the original order index of sensitive data to enhance data privacy without degrading the efficiency of comparison operations over encrypted data.

In summary, the contributions of this study are as follows:

- We introduced three techniques for semi-order preserving encryption (SOPE) for numeric data, which work on top of the OPES.
- We also introduced techniques for retrieving the original order of numerical *plaintext* data from encrypted semi-ordered values.
- We empirically proved the efficiency of the proposed techniques through several experiments.

The rest of this paper is organized as follows. Section 2 reviews related works and previous findings. Section 3 presents the notions and properties of the proposed techniques. Section 4 provides detailed examples and analysis of the proposed techniques. In section 5 we experimentally evaluate and compare the algorithms in the three techniques with the baseline method. Finally, section 6 concludes the paper.

2 Related Work

A number of OPESs have been proposed in the literature [1, 2, 3, 4, 5, 10, 6, 9].

Agrawal *et al.* [1] proposed their OPES as follows: first, they model the input and target distributions as linear splines. Next, they flatten the *plaintext* database into a uniformly distributed database. Furthermore, they transform the uniformly distributed database into a cipher database.

The work by Bebek [2] generates a sequence of random numbers. Moreover, the jth value of the random number is added to the jth integer so that the original order is preserved. The problem of this method is its inefficiency in encrypting values. Moreover, the process does not consider insertion of new data into the database.

Boldyreva *et al.* [3] proposed an order-preserving symmetric encryption. In their construction, they used a natural relation between a random order-preserving function and the hypergeometric probability distribution.

Different from the above schemes, Boldyreva *et al.* [4] considered cryptography based OPES. It first defines the ideal OPES whose encryption function is selected uniformly at random from a set of all strictly increasing functions. Although the ideal OPES is not feasible; it can be used as a security goal for a realistic OPES. They proposed a method to map a *plaintext* x to its *ciphertext* using a binary-search process in the *ciphertext* space and then map back the searched points using hypergeometric distribution to the space.

In the method proposed by Ozsoyoglu *et al.* [10], a sequence of strictly increasing polynomial function is used to construct the OPES. Encrypted value of an integer x is derived from iterative operations of encryption functions on x. The OPES security algorithm is difficult to analyze because it is not constructed using basic formal cryptographic algorithms.

Hacigümüş *et al.* [6] proposed a method that divides the domain of *plaintext* into multiple partitions and then assigns an identification, which can be order preserved, for each partition. They used a mapping function between the *plaintext* and encrypted value. A disadvantage of this encryption algorithm is that it cannot compare all the *plaintexts* (e.g. the *plaintexts* in the same partition).

Zheli Liu *et al.* [9] proposed a method that uses message space expansion and nonlinear space split to hide data distribution and frequency.

Xiao *et al.* [13] developed protocols called "DOPE" and "OE-DOPE", which can realize OPES in multi-user systems. To ensure that no entity in the system knows the OPES encryption key, they introduced a group of key agents into the system and proposed the DOPE protocol.

Ce Yang et al. [14] proposed an SOPE, although with the sacrifice of precision. In our proposed work we maintain the precision of values in the database. Values do not change when we decrypt the order information.

Each of the abovementioned algorithms considers storing the order value to the disk, which is in fact risky. Therefore, we propose a new approach to enhance security, which works on top of the OPES algorithm [1, 3, 4, 9].

Besides OPES, some of the secure multiparty computations use similar techniques, which are shown in Sections 4.1 and 4.2. Among them, Hamada *et al.* proposed a secure multiparty computation of radix sort in [7]. The idea used in their secure computation is the same as in Section 4.2, where we consider the union of other parties' data as an encrypted database. In addition to this idea, we consider general database queries on the encrypted database in this study.

3 Preliminaries

3.1 Types of Attack Considered

- *Ciphertext-only attack*
  In cryptography, a ciphertext-only attack (COA) or known-*ciphertext* attack is an attack model for cryptanalysis, where the attacker is assumed to have access only to a set of *ciphertexts*.

- *Chosen-ciphertext attack*
  A chosen-ciphertext attack (CCA) is an attack model for cryptanalysis, where the cryptanalyst can gather information by obtaining decryptions of the chosen *ciphertexts*. From this information, an adversary can attempt to recover the hidden secret key used for decryption.

- *Known-plaintext attack*
  Known-plaintext attack (KPA) is an attack model for cryptanalysis, where the attacker has access to both the *plaintext* (called a crib), and its encrypted version (*ciphertext*). These can be used to further reveal secret information such as secret keys and code books.

3.2 Threat Model

We adopt the threat model as given below.

- *Storage system used by the database software is vulnerable to compromise.*
  Current database systems usually implement their own storage management. However, the storage system remains as a part of the operating system. An adversary can compromise the storage system by accessing the database files using a path other than through the database software, or by physically acquiring the storage media in the extreme case. Attackers may obtain access to the database file. Since values in the file are encrypted by our proposed method, such an access may initiate a *COA*.

- *The database software is trusted.*
  We assume that the database software is trusted to perform encryption of query constants and decrypt query results. We also assume that some values in the database software's memory may be accessible to adversaries. An adversary can initiate a KPA from these values.

- *All disk-resident data is encrypted.*
  We assume that the database software encrypts schema information such as table and field names; metadata such as column statistics; recovery logs; and data values. Therefore, an adversary is unable to guess the data distribution.

- *The attacker may manage few plaintext values of some chosen ciphertext.*
  With help from the database operator, an attacker can manage the plaintext values of some chosen cipher texts, which are not a part of the original values stored in the database. From such information, the attacker can initiate a CCA.

3.3 Order Preserving Encryption

Encryption is the most effective way to achieve data security. Unencrypted data is termed as *plaintext*, and encrypted data as *ciphertext*. In cryptography, a *key* specifies the particular transformation of *plaintext* into *ciphertext*, or vice versa during decryption.

Let *plaintext* and *ciphertext* be p and c, respectively. Let k_1 and k_2 be the encryption and decryption *keys*. We denote encryption and decryption by the following functions:

$c = encryption(p, k_1)$

$p = decryption(c, k_2)$

If $k_1 = k_2$, then the scheme is known as *symmetric-key cryptography* else *asymmetric-key cryptography.*

Order preserving encryption is somewhat different from ordinary encryption. If we assume that a database P consists of $|P|$ *plaintext* numeric values, which are represented as $P = p_1, p_2, ..., p_{|P|}$, where $p_i < p_{i+1}$, then, when we encrypt the *plaintext* values into *ciphertext* values, which are represented as $\tilde{C} = \tilde{c}_1, \tilde{c}_2, ..., \tilde{c}_{|P|}$, it must be ensured that $\tilde{c}_i < \tilde{c}_{i+1}$ $(i = 1, ...|P| - 1)$. This means that the order of encrypted values must be the same as the *plaintext* values. In our opinion, the order itself is sensitive information. Therefore, we think that data privacy is not ensured enough in the conventional OPES scheme.

3.4 Semi-Order Preserving Encryption

SOPE is an enhancement of the OPES. It is not a standalone scheme; it works on top of the OPES. To enhance the privacy of the order preserved encrypted values in $\tilde{C}$, we map these values to semi-order preserving values. On mapping, $\tilde{C} = \tilde{c}_1, \tilde{c}_2, ..., \tilde{c}_{|P|}$ becomes $C = c_1, c_2, ..., c_{|P|}$. In sequence C, its ordered sequence is perturbed to a new *ciphertext*, in which the order is no longer preserved.

4 Semi-Order Preserving Encryption Techniques

We propose three different techniques in this study that can be integrated on top of the OPES [1, 3, 4, 9] algorithm. Perturbation of *plaintext* is used in the first technique and B-Tree structure for semi-ordering the original order index, in the other two.

Figure 1 illustrates the basic system for integrating our proposed techniques with an encrypted private database and the OPES algorithm. This figure also presents the scopes and the frameworks of proposed techniques.

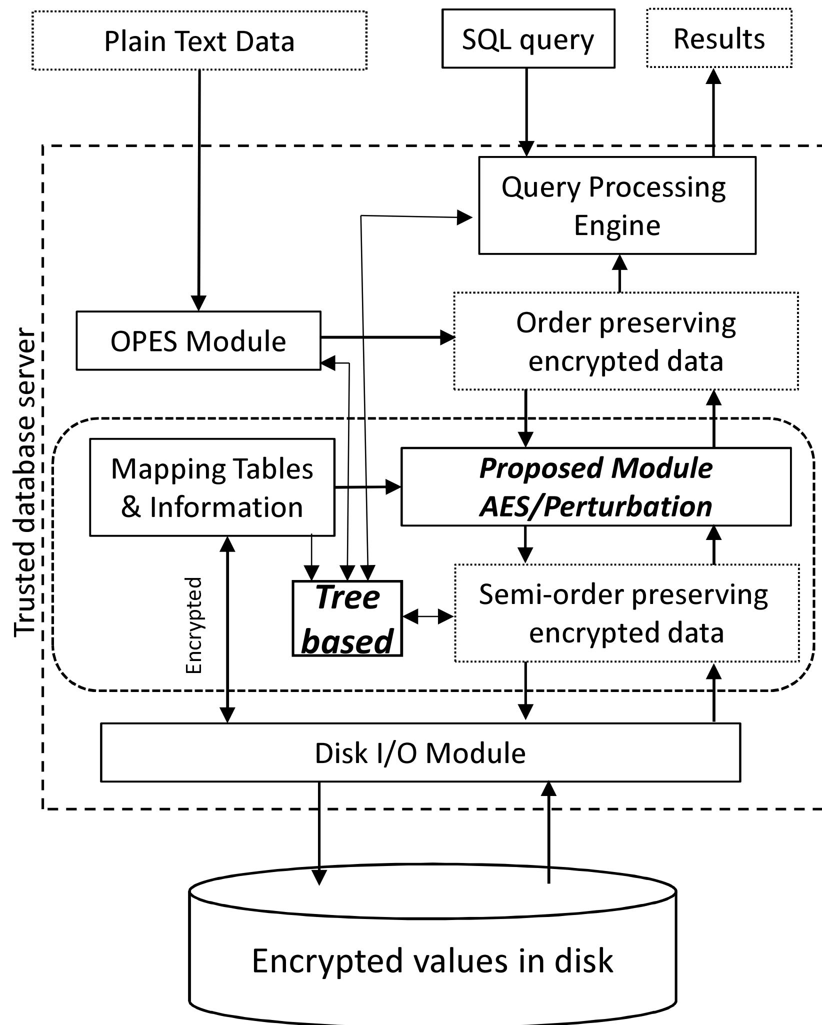

Figure 1: Basic idea of the proposed method

As described in Figure 1, the OPES module encrypts the original numerical *plaintext* values into order preserving encrypted data. However, the order indexes of the original data are sensitive information; therefore, storing the order indexes directly to the disk is also a type of security violation.

In this regard, the techniques proposed in this study take the original order index provided by the OPES module as input and produce semi-order preserving perturbed index as output, which can be stored on physical disk.

4.1 Semi-Order Preserving Technique using OPES and Two-way Perturbation

In the first proposed algorithm, we consider the perturbation of the order information in such a manner that we can easily sort the perturbed values and execute queries efficiently.

For simplicity, we assume that the database with sensitive data consists of a single table with a single column. Let us assume that we have a column of *plaintext* values: $[43, 253, 629, 69, 521]$, and we obtain $[2089, 3458, 7501, 2923, 6303]$ by applying the OPES to the plaintext values.

We use the conventional OPES method for encryption; furthermore, we perturb the values that we obtain from the OPES. To enhance privacy of the encrypted order preserved values in $\tilde{C}$, we map the m-digit *ciphertext* values for each digit and generate perturbed m-digit *ciphertext* values. After the mapping procedure, $\tilde{C} = \tilde{c}_1, \tilde{c}_2, ..., \tilde{c}_{|P|}$, in which, $\tilde{c}_i < \tilde{c}_{i+1}$ becomes $C = c_1, c_2, ..., c_{|P|}$, in which the order is no longer preserved, and we call the sequence: "*perturbed ciphertext values*". The order in $\tilde{C}$ is perturbed in C. Figure 2 shows an example of the perturbation step.

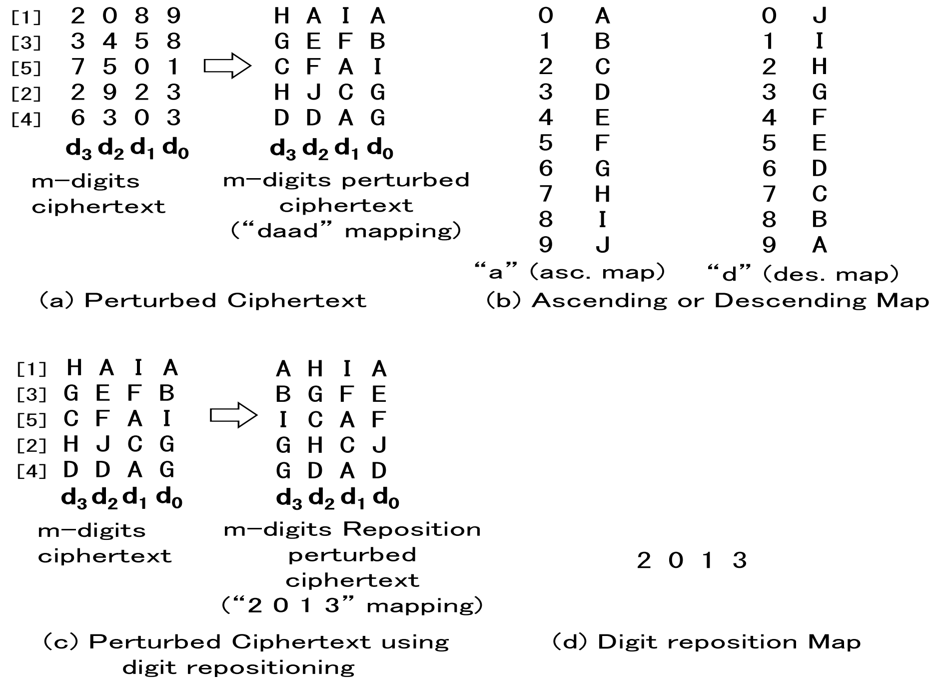

Figure 2: Two-way Perturbation Example

Let $\tilde{c}_i$ be an m-digit *ciphertext* and $\tilde{c}_i[j]$ $(0 \leq j \leq m-1)$ be the jth digit value of $\tilde{c}_i$. We randomly choose one of the two mappings in each digit: "ascending order map" denoted "a" and "descending order map", denoted "d". We map $\tilde{c}_i[j]$ to $c_i[j]$ so that the order becomes $c_{i_1}[j] < c_{i_2}[j]$ $(c_{i_1}[j] > c_{i_2}[j])$ in the ascending (descending) order map if $\tilde{c}_{i_1}[j] < \tilde{c}_{i_2}[j]$ $(i_1 \neq i_2)$

Figure 2 (b) shows an example of the mapping. In the example, "3" is mapped to "D" in ascending order map and to "G" in descending order map. Note that in each digit, the original order in the *ciphertext* is preserved in the ascending order whereas, reversed in the descending.

Figure 2 (a) shows the perturbed *ciphertext* when we randomly chose "daad"; this means that the digits from 3-0 are mapped to "d", "a", "a", and "d" in that order. For example, "2089" is mapped into the perturbed *ciphertext* value "HAIA", in which the 1st value "2" is mapped to the descending order value "H" and the 2nd value "0" is mapped to the ascending order value "A". Similarly, "8" and "9" are mapped to "I" and "A", respectively.

Next, we randomly choose a reposition map and apply repositioning of each digit in a number according to the map. Figure 2 (c) shows the reposition of "HAIA" when we randomly choose "2013" as a reposition map. In the reposition, "H"(the 3rd digit) moves to the 2nd position, the

2nd digit “A” to the 0th position, the 1st digit “I” to the 1st position, and the least significant digit “A”(the 0th digit) to the 3rd position. Finally, after the repositioning, “HAIA” becomes “AHIA”.

The map information such as “daad” and “2013” is stored in the database system; hidden from users. It should be noted that, an adversary cannot infer sensitive data such as map information, number of partitions (digits) each value is partitioned into, number of bits in each partition (digit) , and whether ascending or descending map is used for each partition (digit).

In practice, *ciphertext* values stored in the database are neither decimal nor character values such as the ones in the previous example; they are binary values similar to the example in Figure 3. It should be noted that, in the figure, an OPES output “20171119” (in decimal) is divided into eight digits (each digits has four bits). Each digit is mapped based on the secret mapping “aaddddaa” into mapped value with padded random noise and repositioned with the secret reposition map “03571264.”

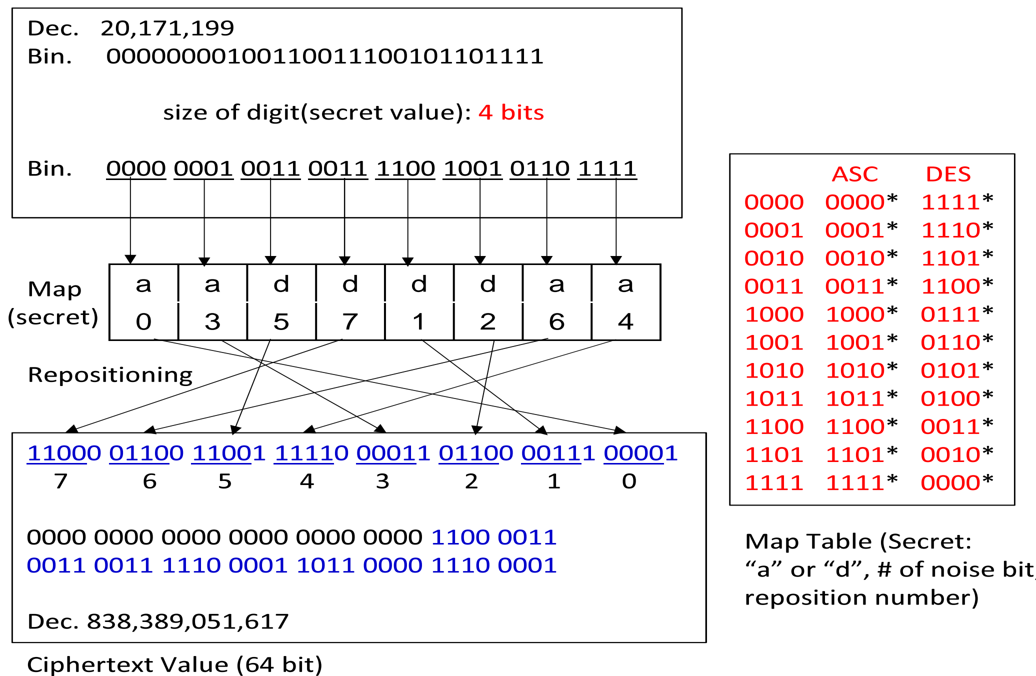

Figure 3: Detailed Binary Example of Two-way Perturbation

Therefore, an adversary cannot infer the original order of the *ciphertext* correctly without knowing the map information.

However, the database system, which has the map information, can compare the perturbed *ciphertext* values. For example, the database system can know that “AHIA” is smaller than “BGFE” because the 2nd digit (reposition of the original most significant digit or 3rd digit) in each number are “H or “G”, where “H” is larger than “G” in descending order.

4.2 Sorting on Perturbed Values

Radix sort can be performed without decrypting data and has complexity of $O(mn)$ where m is the number of digits. Generally, m is suitably small compared with n. Sorting ability of perturbed values includes all the queries in our proposed method that can be executed efficiently in OPES. In perturbed values, order in each digit is preserved and the database system can compare the order in each digit with the help of order and reposition maps. The database system can sort the database by using “radix sort” [11]. We consider the order map “daad” and reposition map “2013.” First, the sort algorithm ordered the values using the 3rd digit (a reposition of original 0th digit) in descending order, which can be done in $O(n)$ where n is the number of data. Preserving the order

of the 3rd digit (a reposition of original 0th digit), it ordered the values using the next digit, i.e., the 1st digit (a reposition of original 1st digit) in ascending order. It continues this procedure until the most significant digit is handled. Figure 4 shows the procedure of radix sort for the example under

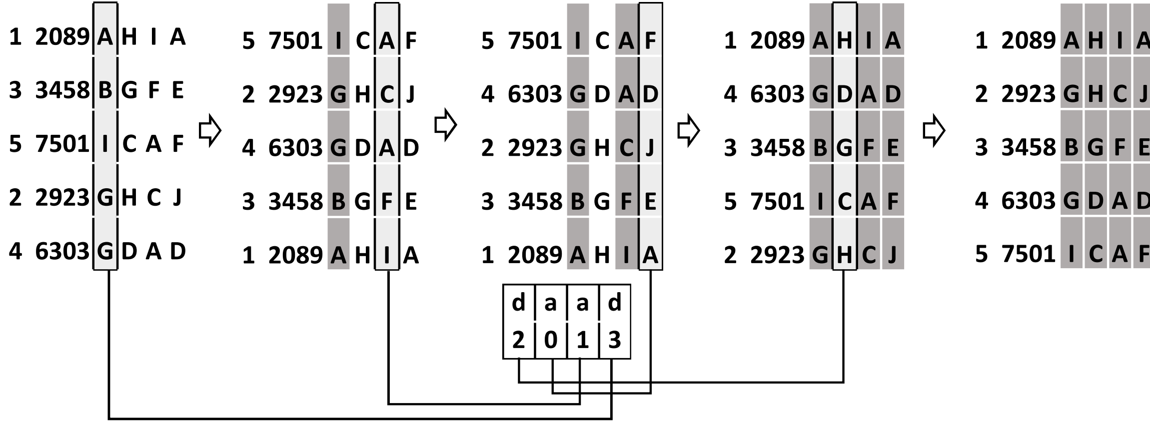

Figure 4: Radix Sort on Perturbed *ciphertext*

consideration. By using the 3rd digit (a reposition of the original 0th digit), the database system sorts the database in descending order; the original sequence {[1],[3],[5],[2],[4]} becomes {[5],[2],[4],[3],[1]}. Next, keeping the order of the 3rd digit, the database is sorted in ascending order of the 1st digit, which is a reposition of the original 1st digit. Both [5] and [4] have the same value in the 1st digit. In this case, order of the previous result is maintained, which produces {[5],[4],[2],[3],[1]}. This procedure is continued to complete the radix sort.

4.3 Query on Perturbed Values

4.3.1 Comparison Operator

SQL uses operators such as = (equal to), <> (not equal to), $>$ (greater than), $<$ (less than), $\geq$ (greater than or equal to) and $\leq$ (less than or equal to) for comparing numerical values. In this section, as shown in Figure 5, we explain how to execute the queries using the abovementioned operators. Here we assume that a user has submitted the following query:

```
SELECT ... WHERE value > 70
```

Conventional OPES detects the probable order of the value, i.e., 70 using its *Lookup Table*. It should be noted that the *Lookup Table* does not include the complete order index but stores the order index of certain sample data and applies the mapping function to compute the order position. Details of this technique can be found in [1]. For the example in our experiment, we consider that the position of 70 in the *Lookup Table* is 2.1, which is between 2 and 3.

Therefore, the *Query Modifier* rewrites the query as follows:

```
SELECT ... WHERE order > 2.1
```

When such a modified query is submitted to a DBMS engine, which has the ordered data, it returns all the tuples that satisfy the order condition. In this manner, the tuples with the order index below 2.1 i.e. {[1], [2]} are eliminated. Thereby, the tuples whose original values are below 70, i.e., {43, 69} are excluded from final the result.

Similarly, queries with other operators can be modified using the query modifier using similar procedures on our proposed techniques.

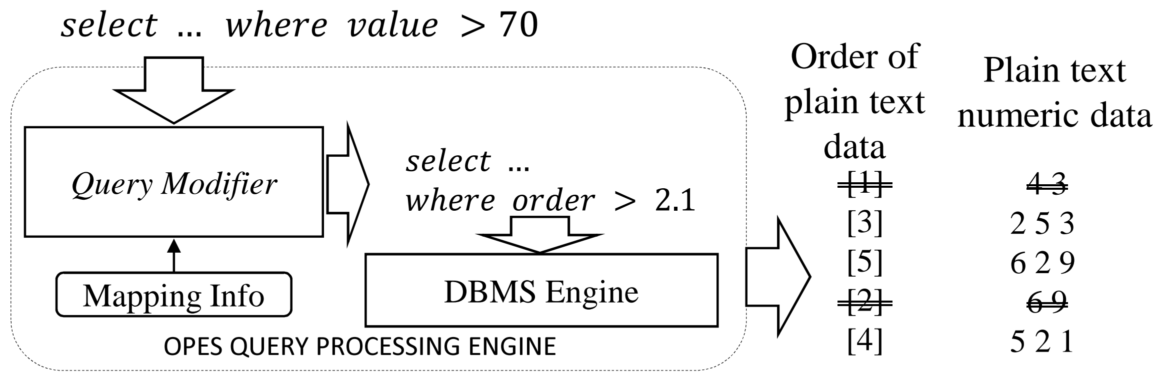

Figure 5: Example of Comparison Operator

4.3.2 Aggregation Function

We now consider Figure 6, and assume that a user has submitted the following query:

```
SELECT MAX(value) ...
```

The query modifier in OPES query processing engine modifies the query as:

```
SELECT MAX(order)...
```

When the modified query is submitted to the DBMS engine, it returns the tuple with the highest value of the order index. Therefore, all tuples except {[5]} are rejected. The value can be retrieved from *plaintext* data (i.e., the tuple with value 629).

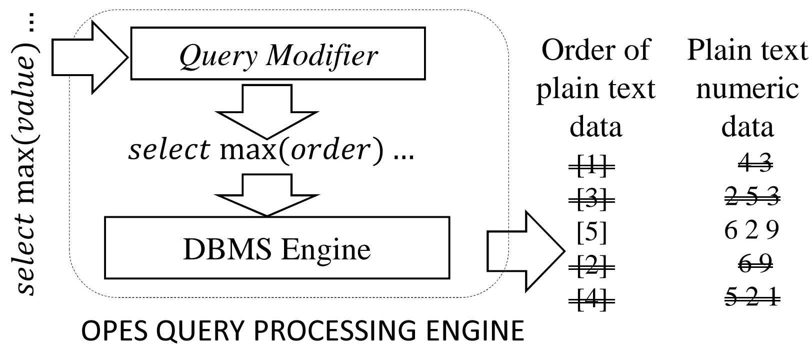

Figure 6: Use of Aggregate Function

However, similar to other conventional order preserving techniques, two aggregation functions: "SUM" and "AVG" cannot be handled without decrypting the original value in the mentioned approach.

4.3.3 Insertion and Deletion Operations

Assume that a user has submitted the following query:

```
INSERT INTO ... VALUES (289)
```

Whenever such an operator is submitted to order preserving techniques, the *Query Modifier* determines the probable order of new data value. Considering the example in Figure 7, and the sample data of our running example in Figure 2,

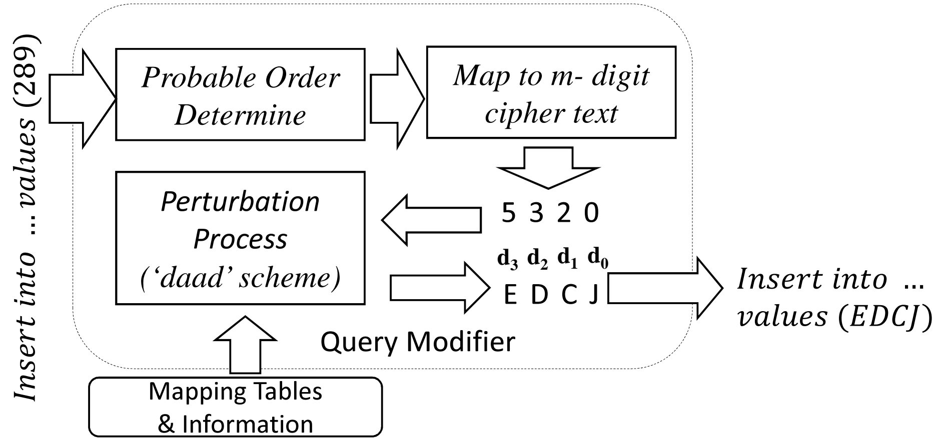

Figure 7: Insertion Operation

it is obvious that the probable order of the new data value 289 is in between the 3rd and 4th position. Therefore, the order preserved m-digit *ciphertext* value would lie between 3458 and 6303. Let us assume that the m-digit order preserved *ciphertext* value of 289 is 5320. The process of such mapping is discussed in detail in [1]. After obtaining value 5320, we use our semi-order preserved scheme to determine the encrypted value. The perturbation process described in Section 4.1, maps the m-digit order preserved *ciphertext* to a perturbed semi-order preserved *ciphertext*. For example, 5320 will be perturbed to $EDCJ$ as per descending mapping table. Let "E" and "J" be the corresponding values for 5 and 0, respectively. Furthermore, let "D" and "C" correspond to values 3 and 2, respectively, in the ascending mapping table. Thus, the operation will be modified to:

```
INSERT INTO ... VALUES(EDCJ)
```

and submitted to a database engine. Deletion operation is usually performed using comparison operators. Suppose that a user submits an operation:

```
DELETE ... FROM ... WHERE (VALUE > 70)
```

The process is almost identical to that expressed in Figure 5. The *Query Modifier* has to determine the probable order of 70 and modify the operation accordingly. The operation is transformed to the following after the modification:

```
DELETE ... FROM ... WHERE (ORDER > 2.1)
```

If there is no *where* clause, the *delete...* operation can be submitted to the database engine directly.

4.4 Semi-Order Preserving Technique using Block-wise OPES using Tree

In this method, we first construct *plaintext* intervals using Algorithm 1, and further using the intervals, construct a B-tree in *plaintext* domain as shown in (1) in Figure 8. Let N be the number of leaves of the B-tree. Next, we split the *ciphertext* domain into N disjoint intervals as shown in (2) in the figure. Furthermore, we assign one of the disjoint intervals at random for each leaf of the B-tree as shown in (3) in the figure. Based on the B-tree, we apply the conventional OPES for each leaf as shown in (4) in the figure. For example, *plaintext* values $[12, 25)$ are encrypted to values in $[100, 199]$ so that the original order is preserved. Figure 9 shows examples that are encrypted by the tree in Figure 8. *Plaintext* value 14 reaches to leaf $[12, 25)$ and is encrypted to a value in $[100, 199]$. Similarly, *plaintext* value 20 is also encrypted to a value in $[100, 199]$. In the example, 14 and 20 are encrypted to 121 and 171, respectively. Note that the order in *plaintext* domain is preserved in the *ciphertext* domain. On the contrary, for two *plaintext* values belonging to different leaves, the order is not preserved, which means that the order privacy is preserved.

4.4.1 Interval Choosing Algorithm

Our goal is to hide the sensitive order information from the adversary. Therefore, choosing an interval is critical in *plaintext* domain. From the above example in Figure 8, it is obvious that the orders in the same block pair of (*plaintext* and *ciphertext*) are identical. Therefore, the goal of this algorithm is to avoid two or more order sensitive values that reside in the same block. The algorithm is given below

Result: Intervals for plaintext domain
Take all the plaintext values in the plaintext domain;
ListA=First value in plaintext domain;
while *all plaintext values are not processed* **do**
  if *Is the current value order sensitive to values in ListA* **then**
    Assign new plaintext interval for ListA values and store the interval;
    empty ListA;
    ListA=Current Value;
  else
    ListA=ListA + Current Value;
  end
end

Algorithm 1: Interval Choosing Algorithm

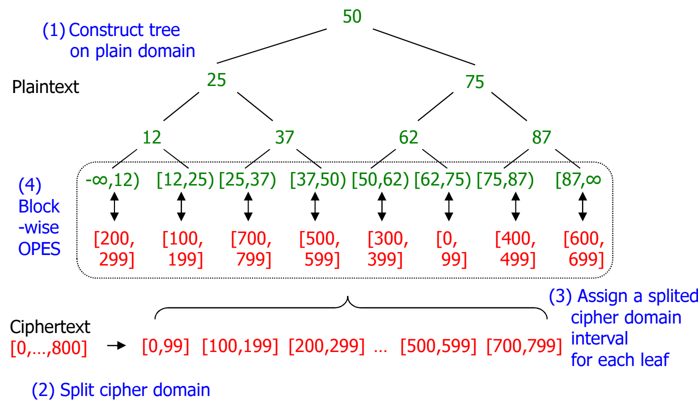

Figure 8: Block-wise Semi-Order Preserving Encryption using Tree

Plaintext	3	10	14	20	28	35	39	46	52	58	65	71	77	83	91	96
Ciphertext	215	245	121	171	733	772	513	562	316	361	25	65	411	433	622	651

Figure 9: Example of Block-wise OPES using Tree

4.5 Semi-Order Preserving Technique using Dynamic Block-wise OPES using Tree

This is a variation of the technique described in Section 4.4. It can handle two things. Firstly, when we insert an order sensitive value in an interval, it dynamically splits the interval and inserts the value into the non-sensitive split. For example, values in interval 50-62 go to the interval 300-399 and maintain the same order. If 56 is order sensitive to some values in the interval 50-55, the interval 50-62[300-399] will split into 50-55[300-399] and 56-62[1500-1600] as shown in Figure 10. Value 56 is converted to *ciphertext* using [1500, K5, 1600] and inserted into the database.

Secondly, it is known that in some databases, operation hours may have peak and off-peak hours. During peak hours, the database is considerably busy; however, during off-peak hours, it is relatively

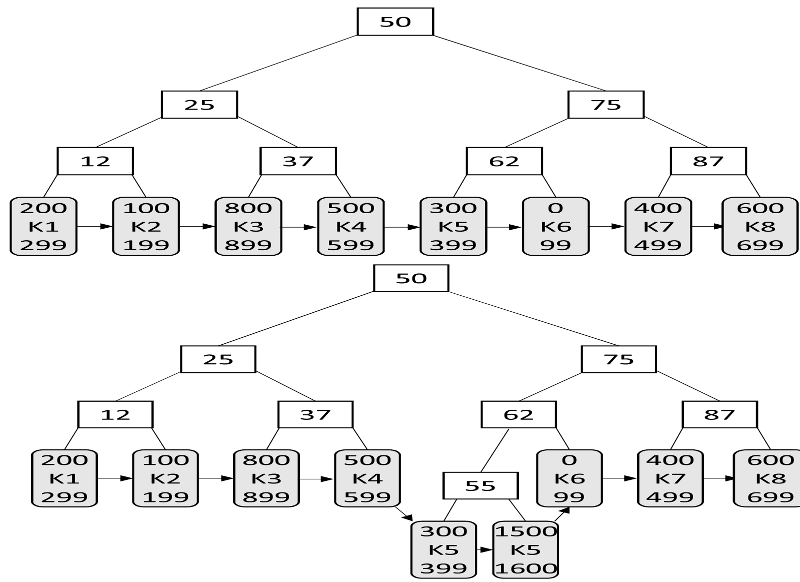

Figure 10: Dynamic Split of Interval

less busy. So, during off-peak hours, we can apply the dynamic change of intervals and avoid the adversary's continuous effort to disclose interval pairs (*plaintext*, *ciphertext*). Figure 11 shows that

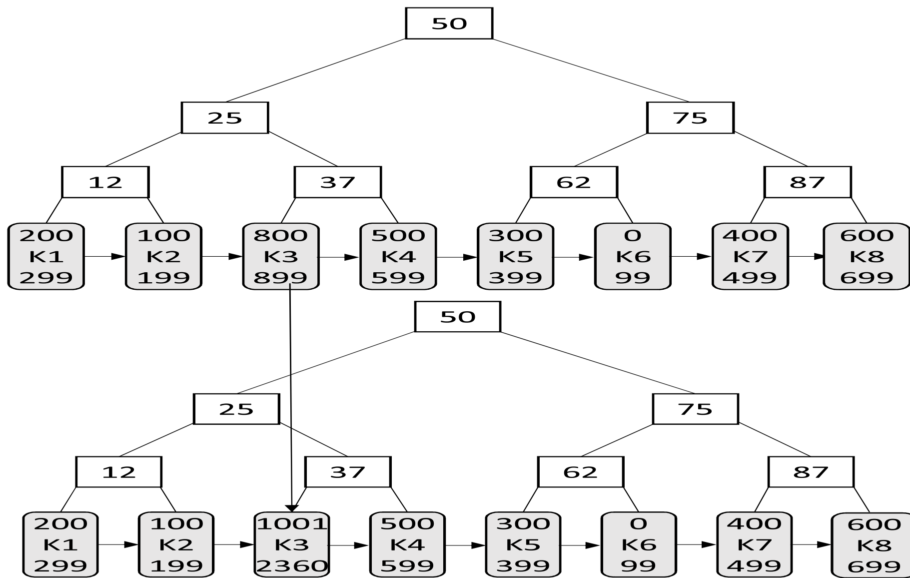

Figure 11: Dynamic Interval Update in "off-peak" Operation Hours

during the off-peak hours, the interval 800-899 changes to 1001-2360 and all values in the database in the interval 800-899 are converted into interval 1001-2360. When there are static interval pairs, an adversary can collect the intervals from internal sources. However, in the case of dynamic intervals, one-time disclosure of interval pairs does not affect the security.

4.6 Query on Block-wise OPES Values by Tree

4.6.1 Comparison Operator

We can efficiently execute a query using comparison operators: $=, <>, <, >, >=, <=$. Suppose that we have to execute a query:

```
SELECT ... FROM TABLE A WHERE VALUE > 70
```

We assume that 70 is a *plaintext* value, and Table A contains encrypted values. We aim to execute this query in encrypted values directly and obtain the query results in encrypted form. When we observe the encryption tree from the left to right leaves, the *ciphertext* intervals are unordered; however, their corresponding *plaintext* intervals are ordered. When we traverse the tree from left to right in *ciphertext* interval, it means that we are traversing it in ascending order for corresponding *plaintext* intervals. For the query execution, we searched the tree with value 70, and reached the leaf $[0, K5, 100]$. We use key $K5$ and interval $[0, 99]$ to encrypt the value 70. After encryption, *plaintext* 70 becomes *ciphertext* 55. Values greater than 55 in the [0,100] and all values in the intervals [400,499] and [600,699], which are on the right side of [0,99] in the tree, are actually the result of the query in encrypted form.

4.6.2 Aggregation Function

We know that the leaves of the encryption tree are in ascending order in *plaintext* intervals from left to right; therefore, the leftmost interval contains MIN value and rightmost contains MAX value. This is also true in *ciphertext* intervals. MIN is the minimum value in the leftmost *ciphertext* interval and MAX is the maximum value in the rightmost *ciphertext* interval in the tree.

Therefore, this method can efficiently handle the aggregation function of `MIN` and `MAX`. It is obvious from Figure 8 that `MIN` is the minimum of values within the leftmost node interval (minimum of values within [200, 299]) and `MAX` is the maximum of values within the rightmost node interval (maximum of values within [600, 699]).

4.6.3 Insertion and Deletion Operations:

Suppose, an insertion operation

```
INSERT INTO ... VALUES(16)
```

is submitted. For executing the above query, our proposed method searches the tree using value 16 and finds the leaf-node $[100, K2, 199]$. Furthermore, by using K2 and the interval $[100, 199]$, 16 is encrypted into 150 and inserted into the database.

Consider that a delete operation:

```
DELETE ... FROM ... WHERE (VALUE>70)
```

is submitted. We know that the intervals in the leaves of the encryption tree are in ascending order in *plaintext* domain. Therefore, deleting values greater than 70 means deleting values higher than 70 in the intervals that contain 70 and all values in the intervals to its right. This is also true in *ciphertext* intervals.

Our proposed method finds the leaf for value 70 in the tree, which comes out to be $[0, K6, 99]$. Therefore, 70 is encrypted as 50 by using K6 in the interval $[0, 99]$. The above query deletes all encrypted values greater than 50 in the interval $[0, 99]$ as well as in the intervals $[400, 499]$ and $[600, 699]$, which are on the right side of $[0, 99]$ in the tree.

4.7 Evaluating SOPE Against Threat Model

We build our system on top of the OPES. We use the results of [1, 3, 4, 9] for the provable security. So, the system has all the security aspects of [1, 3, 4, 9] and additionally hides the order.

4.7.1 Semi-Order Preserving Technique using OPES and Two-way Perturbation

The adversary does not have access to the OPES encryption key, perturbed mapping, repositioning, and noise bits. If an adversary wants to know the order information by brute force attack, he has to determine the partitions, positions of noise bits, ascending-descending maps, and the reposition map. For n digit binary numbers, the partitions can be done in 2^{n-1} ways. If we consider k noise bits, they can occur in ${}^{n}P_k$ ways in the number, and repositioning of the digits can be done in $!n$ ways. Among $2^{n-1} * {}^{n}P_k * !n$ ways, the adversary can guess the correct way only once. Therefore, probability of guessing the correct order through brute force attack is $1/(2^{n-1} * {}^{n}P_k * !n)$.

4.7.2 Semi-Order Preserving Technique using Block-wise OPES using Tree

To guess the exact order, the adversary needs to divide the encrypted values into correct intervals and keys for each of the intervals. If there are N possible values in the database domain, there are 2^{N-1} possible intervals. The probability of guessing the correct interval in brute force attack is $1/(2^{N-1})$. Therefore, it is very difficult to guess the exact intervals. Each interval is encrypted with a different key. An adversary needs both, the OPES key and exact intervals. To ensure extra security, the interval generation algorithm ensures that no two sensitive values fall in the same interval.

4.7.3 Semi-Order Preserving Technique using Dynamic Block-wise OPES using Tree

The recognition of exact interval is very difficult as described in the previous section; however, the adversary may manage intervals from internal sources. In this method, the intervals are dynamically changed over time. Therefore, onetime gathering of the intervals and keys does not help the adversary to break the order information.

4.8 Comparison of the Three Proposed Methods with Respect to Different Types of Attack

4.8.1 Ciphertext-only attack

In this mode of attack, the adversary knows only a portion of the ciphertext but no other statistical information from the database. The OPES (whose security is already proved in [1, 3, 4, 9]) system can withstand this type of attack. As our system runs on the top of the OPES, it can offer protection from COA.

4.8.2 Chosen-ciphertext attack

The adversary can gather information by obtaining the decryption of chosen cipher-texts. From this information, the adversary can attempt to recover the hidden secret key used for decryption. In case of OPE, it is possible to guess the correct key. Our method runs on top of the OPES. Therefore, in this case, *plaintext* is the original value and we obtain *ciphertext* by encrypting *plaintext* firstly by OPES and then by our method($plaintext \rightarrow OPES \rightarrow OPES(cipher) \rightarrow SOPE \rightarrow ciphertext$). Our method hides the OPES(cipher), which is ciphertext for OPES and plain-text for SOPE. Therefore, OPES security described in [1, 3, 4, 9]) can be applied to this situation. In this situation, the adversary can only perform brute force attack, whereas in the tree-based method, they need to find the intervals and then the key. From the earlier section, we know that the probability to guess the correct intervals is $1/(2^{N-1})$, where N is the number of possible values in the database domain. If we have 1,000,000 possible values in a database domain, the probability of detecting correct intervals is $1/(2^{1000000})$. In practical scenarios, the number of possible values in the database domain is much higher.

4.8.3 Known-plaintext attack

In our method, WHICH runs on top of the OPES, *plaintext* is the original value and we obtain *ciphertext* by encrypting *plaintext* firstly by OPES and then by our method($plaintext \rightarrow OPES \rightarrow$

$OPES(cipher) \rightarrow SOPE \rightarrow ciphertext$). Here, the attacker has access to both the plaintext and its encrypted version (ciphertext). However, OPES(cipher) that is *plaintext* for SOPE is hidden. Therefore, our method breaks the connection between *ciphertext* and *plaintext*. The security in [1, 3, 4, 9]) can be applied to this situation.

Our system can run on top of any OPES system. Whenever a researcher finds a new OPES system that is more secure and efficient than previous ones, our system can run on top of the new system and hide the order information.

5 Experiments

We conducted a series of experiments to evaluate the effectiveness and efficiency of the proposed methods. The proposed algorithm was implemented using *Matlab R2016b*. We conducted experiments on a PC with fourth-generation Intel® Core™i7 processor, 3.4 GHz CPU, and 8 GB main memory, running on 64-bit Microsoft Windows 10 Enterprise edition operating system. Each experiment was repeated five times, and averaged. In our experiments, we used five datasets: 10k, 100k, 500k, 1m, and 10m, each of which consists of one field of the randomly generated integer. Datasets 10k, 100k, 500k, 1m, and 10m contain 10,000, 100,000, 500,000, 1,000,000, and 10,000,000 records respectively.

Hiding of order can be done by encrypting the output of OPES values. AES [8] is a well-known encryption technique. We can hide the order information by applying AES to the output of OPES. We consider this as a baseline technique to compare the three proposed methods.

5.1 Comparison Operator

In this section we consider query of the type: *"select * from Table A where A.x > value1"*. Here, table A consists of 10M records. Time is calculated for the number of records returned by the select query. Figure 12 shows that the three proposed methods can efficiently handle the comparison operator in the execution of select query over encrypted values, compared to the AES based baseline method. The tree-based method is more efficient and secure than the others. From the figure, it can be seen that the perturbation based method is also efficient in comparison to the AES based method.

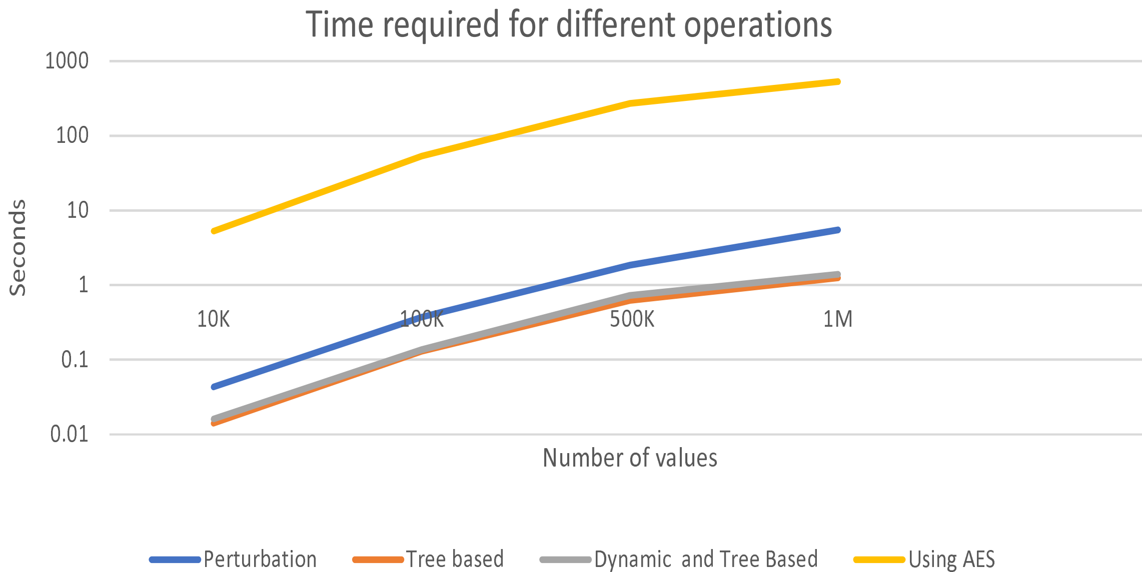

Figure 12: Time required for different operations

5.2 Order Hiding

Figure 13 shows execution times required for order hiding using the baseline AES, perturbation, tree-based, and dynamic tree-based methods. We varied the data size from 10,000 to 1,000,000 and saw that execution time was within acceptable range. The complexity of order hiding in the perturbation method is $2O(n)$, whereas, it is $O(n)+log(n)$(searching the tree $log(n)$ + linked leaves traversal $O(n)$)for the tree-based method. The figure shows that the execution time increases linearly with data volume. The figure also clearly shows that the three proposed methods are very efficient compared to the AES based method. Dynamic tree-based method required slightly more time than the tree based method; however, it made the encryption more secure.

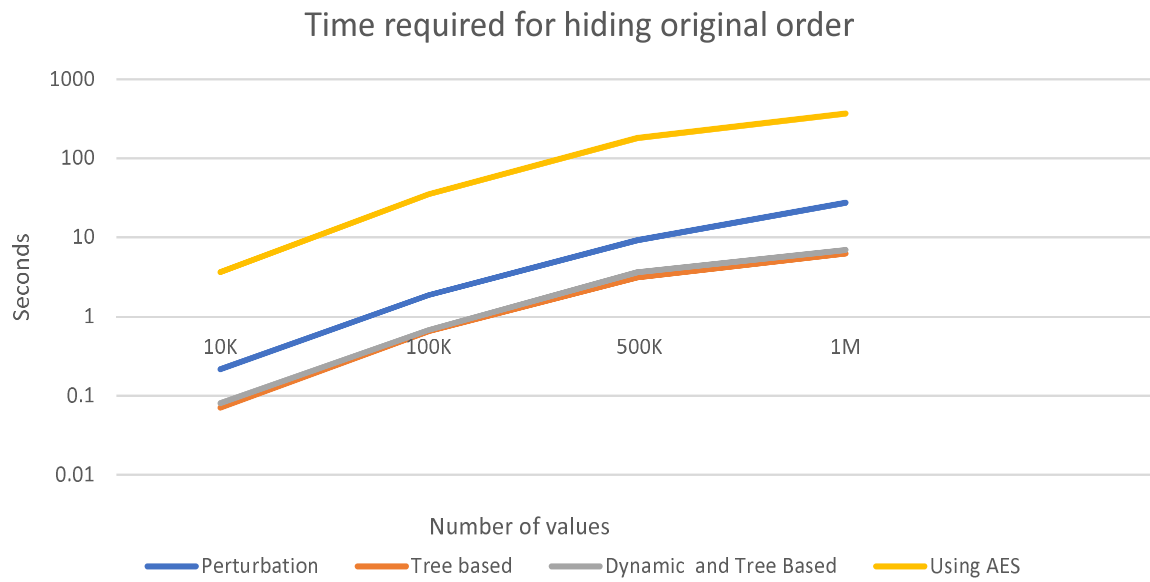

Figure 13: Total Time for Order Hiding

5.3 Retrieving of the Original Order

Figure 14 displays the total time required for retrieving the original order. We varied the length of OPES output data volume from 10,000 to 1,000,000.

Figure 14 shows the time required for retrieving the original order with respect to different data volumes. From the figure, we observe that tree-based and dynamic tree-based methods are faster than others. The running time of perturbation based method is acceptable compared to the baseline AES based method. In case of perturbation based method radix sort is required, therefore

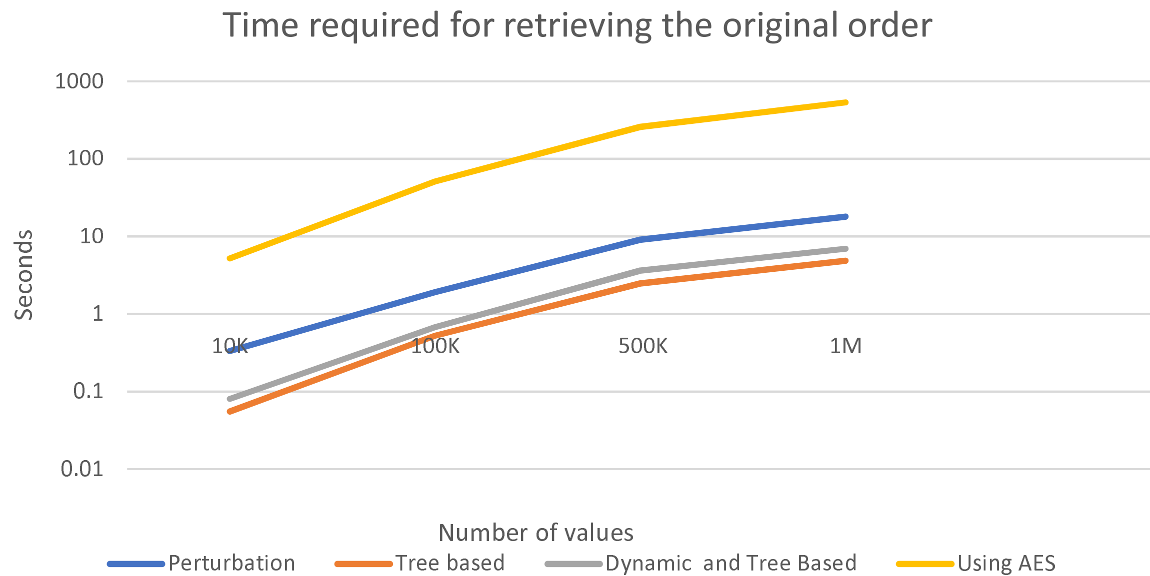

Figure 14: Time required for retrieving the original order.

its complexity becomes $O(mn)$, where n is the number of data and m is the number of digits in a number. The tree-based method needs binary search and linked list traversal, having complexity of $O(n) + log(n)$. In case of ordering the unordered data, AES has considerable time overhead.

The figure shows that performances of our three proposed methods are acceptable regarding time complexity.

6 Conclusion

As discussed in the Introduction section, there are many situations in which we do not want to reveal order information. In such cases, our proposed encryption technique successfully hides the order information. It also provides an extra protection layer to the OPES method. Execution time for the proposed methods is acceptable as shown by experiments. We can execute the queries using different comparison operators without decrypting the original value. Therefore, our technique is suitable for practical use.

So far, we have confirmed the feasibility of the proposed schemes. We are also considering distributed computation of the comparison operators of the proposed schemes in the MapReduce framework.

Acknowledgments

This work is supported by KAKENHI (16K00155,17H01823) Japan. Saleh Ahmed is supported by the Japanese Government's MEXT Scholarship.

References

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order-preserving encryption for numeric data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 563–574, 2004.

[2] G. Bebek. Anti-tamper database research: Inference control techniques. *Technical Report EECS 433 Final Report, Case Western Reserve University*, 433, 2002.

[3] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[4] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. *Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[5] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. Practical order-revealing encryption with limited leakage. In *Fast Software Encryption (FSE)*, 2016.

[6] H. Hacigümüş, B.R. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the databaseservice-provider model. *Proc. of the ACM SIGMOD Conf. on Management of Data, Madison,Wisconsin*, pages 216–227, 2002.

[7] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. *IACR Cryptology ePrint Archive*, 2014:121, 2014.

[8] Vincent Rijmen Joan Daemen. *The Design of Rijndael AES — The Advanced Encryption Standard*. Springer International Publishing, 2002.

[9] Zheli Liu, Xiaofeng Chen, Jun Yang, Chunfu Jia, and Ilsun You. New order preserving encryption model for outsourced databases in cloud environments. *Journal of Network and Computer Applications*, 59:198 – 207, 2016.

[10] G Ozsoyoglu, D Singer, and SS Chung. Anti-tamper databases: Querying encrypted databases. *Proc. of the 17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security, Estes Park, Colorado*, pages 133–146, 2006.

[11] Anthony Vinay Kumar S and A. Arya. Fastbit-radix sort: Optimized version of radix sort. In *2016 11th International Conference on Computer Engineering Systems (ICCES)*, pages 305–312, 2016.

[12] Douglas R. Stinson. *Cryptography : theory and practice*. Chapman & Hall/CRC, Boca Raton, 2005.

[13] Liangliang Xiao, I-Ling Yen, and Dung T Huynh. Extending order preserving encryption for multi-user systems. *IACR Cryptology ePrint Archive*, 2012:192, 2012.

[14] Ce Yang, Weiming Zhang, and Nenghai Yu. Semi-order preserving encryption. *Information Sciences*, 387:266 – 279, 2017.