# Frequency-Hiding Order-Preserving Encryption with Small Client Storage

Dongjie Li[1,2], Siyi Lv[1,2], Yanyu Huang[1,2], Yijing Liu[1,2], Tong Li[1,2,*], Zheli Liu[1,2,*], Liang Guo[3]
College of Cyber Science,Nankai University, China [1]
Tianjin Key Laboratory of Network and Data Security Technology, Nankai University, China [2]
Huawei Technology Co. , Ltd, Shenzhen, China[3]
{lidj,siyilv,huangyy,liuyijing}@mail.nankai.edu.cn,{tongli,liuzheli}@nankai.edu.cn
blue.guo@huawei.com

## ABSTRACT

The range query on encrypted databases is usually implemented using the order-preserving encryption (OPE) technique which preserves the order of plaintexts. Since the frequency leakage of plaintexts makes OPE vulnerable to frequency-analyzing attacks, some frequency-hiding order-preserving encryption (FH-OPE) schemes are proposed. However, existing FH-OPE schemes require either the large client storage of size $O(n)$ or $O(\log n)$ rounds of interactions for each query, where $n$ is the total number of plaintexts. To this end, we propose a FH-OPE scheme that achieves the small client storage without additional client-server interactions. In detail, our scheme achieves $O(N)$ client storage and 1 interaction per query, where $N$ is the number of distinct plaintexts and $N \leq n$. Especially, our scheme has a remarkable performance when $N << n$. Moreover, we design a new coding tree for producing the order-preserving encoding which indicates the order of each ciphertext in the database. The coding strategy of our coding tree ensures that encodings update in the low frequency when inserting new ciphertexts. Experimental results show that the single round interaction and low-frequency encoding updates make our scheme more efficient than previous FH-OPE schemes.

**PVLDB Reference Format:**
Dongjie Li, Siyi Lv, Yanyu Huang, Yijing Liu, Tong Li, Zheli Liu, and Liang Guo. Frequency-Hiding Order-Preserving Encryption with Small Client Storage. PVLDB, 14(13): 3295 - 3307, 2021.
doi:10.14778/3484224.3484228

## 1 INTRODUCTION

At present, encrypted databases (e.g., CryptDB [31], Arx [29], etc) have attracted widespread attention from academia and industry. The range query, one of the popular search operations in encrypted databases, is usually implemented using an encryption primitive called order-preserving encryption (OPE) which preserves the order of plaintexts. The minimum security requirement of OPE is not to reveal any other information besides the ciphertext order. In recent years, many studies [2, 15, 28] have pointed out that this basic security of OPE is insufficient to protect data privacy, since deterministic OPE ciphertexts will leak the frequency of plaintexts. Based on frequency information, the attacker can distinguish the plaintexts using some auxiliary information, such that many attacks (inference attacks [2, 15, 28], volume attacks [14, 16, 19], etc.) successfully reveal the ciphertext or steal other useful information without decrypting the corresponding ciphertexts. Therefore, hiding the frequency becomes a practical security goal when OPE is applied in encrypted databases. In order to reduce information leakage, some order-revealing encryption (ORE) schemes [6–8, 12, 17, 24] have been proposed. Unlike OPE, the ciphertexts of ORE will not reveal the order of the plaintext until the comparison is performed. However, its ciphertexts still cannot hide frequency information.

To hide the frequency, the OPE scheme must not only preserve orders of ciphertexts, but also randomly generate different (indeterministic) ciphertexts/orders for same plaintexts. For example, the plaintexts 4, 5, 5, and 5 are given ciphertexts/orders 112, 113, 115, and 114, respectively, and each ciphertext/order of value 5 is still larger than the ciphertext/order of value 4. Such ciphertexts/orders should be organized by a *state*, otherwise the scheme without the state cannot even achieve the security [30].

Some frequency-hiding order-preserving encryption (FH-OPE) schemes [20, 33] have been proposed. According to the state, the existing FH-OPE solutions can be divided into two categories: client-state schemes and server-state schemes. The typical solution with the client state is Kerschbaum's scheme [20], where the client maintains a tree-structure state mapping plaintexts to OPE ciphertexts. Similar to BCLO [4], it can be theoretically applied in the encrypted database, since it stores all indeterministic OPE ciphertexts on the client. However, this client storage is too large and linear with the number of all records in the database, which is infeasible for actual database applications. The typical solution with the server state is the partial order-preserving encoding (POPE) scheme [33] which stores ciphertexts through a B+-tree-structure state on the server side. The scheme is partial order-preserving since ciphertexts are sorted until any query reaches. When querying, a part of unsorted ciphertexts will be relocated into the appropriate position via interactive comparisons. Although the client needs to remember nothing, each comparison still requires the client to download the compared data, which leads to many rounds of interactions between the client and the server. These additional interactions slow query operations greatly and thus make POPE difficult to be deployed in the encrypted database.

**Motivation**. One fact is that we need to provide the frequency-hiding security for the field which assigns a record the value from a small domain and contains a large number of records (because the

*Zheli Liu and Tong Li are the corresponding authors.


Proceedings of the VLDB Endowment, Vol. 14, No. 13 ISSN 2150-8097.
doi:10.14778/3484224.3484228

**Table 1: Comparison with Previous OPE Schemes. $\lambda$ is denoted as the security parameter and $\epsilon$ is denoted as a constant greater than 0 and less than 1. POPE [33] is set to perform $O(n^{1-\epsilon})$ range queries.**

| OPE scheme | Security | | Interaction | | Client Storage | | Incomparable Elements | Server Storage |
|---|---|---|---|---|---|---|---|---|
| | IND-OCPA | IND-FAOCPA | Insert | Query | Working | Persistent | | |
| BCLO [4] | No | No | 1 | 1 | $O(1)$ | $O(1)$ | 0 | $O(n)$ |
| mOPE [30] | Yes | No | $O(\log n)$ | $O(\log n)$ | $O(1)$ | $O(1)$ | 0 | $O(n)$ |
| FH-OPE [20] | Yes | Yes | 1 | 1 | $O(n)$ | $O(n)$ | 0 | $O(n)$ |
| POPE [33] | Yes | Yes | 1 | $O(\log n)$ | $O(1)$ | $O(n^{\epsilon})$ | $\Omega(\frac{n^{2-\epsilon}}{m} - n)$ | $O(n)$ |
| Our scheme | Yes | Yes | 1 | 1 | $O(N)$ | $O(N)$ | 0 | $O(n)$ |

frequency attacks are easy to succeed on this field even its records are encrypted). Take the field "age" as an example. There are about 100 values on the domain of "age", but there are usually thousands of database records on "age". However, using existing schemes to encrypt records on this field will either cost the client storage linear with the number of all records or frequent client-server interactions. Based on this consideration, we focus on how to design a FH-OPE scheme that avoids the above shortcomings of existing schemes. Moreover, a FH-OPE solution is supposed to provide a feasible way for implementing OPE in the encrypted database as well as applying BCLO [4] to CryptDB [31]. In more details, it only needs to add a few functions to complete the corresponding operations on the database rather than change the database driver.

**Our work**. We propose a new FH-OPE scheme which achieves the small client storage and only costs single round interaction per query. Different from existing works, our scheme is a client-server-state scheme, where the client state is a *local table* and the server state is a tree for organizing ciphertexts in order. The local table maps a plaintext to the number of its ciphertexts stored on the server side and thus its maximum size is equal to the number of distinct plaintexts. When encrypting a target plaintext to the ciphertext, the local table helps the client carry out the server-side position where the ciphertext will be located. Therefore, after receiving the ciphertext along with its position information, the server can directly insert the ciphertext into the server state without any further interaction with the client.

There is a critical challenge for constructing an appropriate server state suitable for the small local table. On the one hand, the ciphertext-comparison operation of existing server-state OPE schemes, such as mutable order-preserving encryption (mOPE) scheme [30] and POPE, requires additional client-server interactions. On the other hand, the tree-structure state of these schemes will trigger a large number of updates on order-preserving encodings/ciphertexts when the tree needs to be rebalanced, which introduces much communicational or computational overhead. To tackle this challenge, we design a server state called *coding tree*, which unbinds the relationship between the encoding/ciphertext update and the tree rebalancing. Unlike the previous works, the rebalancing of the tree in our FH-OPE scheme will never trigger the update of the existing encodings/ciphertexts, such that the frequency of update operations is greatly reduced.

Our contributions are summarized as follows:

- We propose the first FH-OPE scheme with the small client storage and no additional client-server interaction. Table 1 shows the comparison between our work and previous OPE schemes. Our scheme achieves $O(N)$ client storage and is lower than Kerschbaum's scheme [20] whose complexity is $O(n)$, where $N$ is the number of distinct plaintexts and $n$ is the total number of plaintexts. Our scheme achieves single round interaction for each insertion/query, while mOPE [30] and POPE [33] require $O(\log n)$ interactions at least.
- We propose a new *coding tree* with the coding strategy. Our coding strategy greatly reduces the frequency of encoding updates. Experiments demonstrate that the frequency of updates in our scheme is less than that in the previous schemes [20, 30] when inserting the same number of plaintexts.
- We deploy our scheme in the real-world encrypted MySQL database by implementing user-defined functions (UDFs). Experimental results on the implementation highlight our scheme's advantages on the client storage, the recoding frequency, and the client-server interaction. The evaluation demonstrates that our scheme is practical and outperforms previous FH-OPE schemes [20, 33] due to the low recoding frequency and the less interaction.

## 2 RELATED WORK

### 2.1 OPE and Frequency Attacks

Up to now, many researches on OPE/ORE [1, 4–6, 10, 18, 20–22, 25–27, 30, 33] have been proposed, and some of them have been integrated into the database including CryptDB [31], Arx [29], ZeroDB [11], and EncDBDB [13]. The survey [3] systematically summarized previous OPE solutions and evaluated the performances of them.

**OPE and stateful OPE**. According to the methods for preserving the orders of ciphertexts, the OPE schemes are mainly divided into two types: *stateful* OPE and *stateless* OPE. The stateful solution contains a state that dynamically organizes ciphertexts in order, while the stateless one does not. Such a state can be seen as a part of "ciphertexts" which is mutable and necessary to achieve the ideal security, i.e., indistinguishability under the ordered chosen plaintext attack (IND-OCPA). That means a ciphertext does not reveal any other information besides its plaintext's order. Boldyreva et al. [4] has proved that it is impossible for OPE scheme to achieve IND-OCPA with non-mutable ciphertexts. Then, Popa et al. [30] presented an interactive OPE scheme that achieves IND-OCPA via the server state. Due to the security, researches on OPE focused more on the stateful solution in the last decade.

**Frequency attacks**. As mentioned in Section 1, deterministic OPE ciphertexts leak the frequency of plaintexts and thus provide a

way for recovering plaintexts from ciphertexts without decryption. Many attacks [2, 14–16, 19, 28] based on the frequency information are proposed. The cumulative attack [28] tried to match a ciphertext with auxiliary data (which is available for the public) by learning frequency and order information of the ciphertext. It can recover 80% of plaintexts with frequency information. Grubbs et al. [15] proposed a non-crossing attack using bipartite graph non-crossing matching to enforce the consistency of a plaintext-ciphertext sequence. Bindschaedler et al. [2] proposed a Bayesian inference attack which greatly improved the theoretical basis and experimental effect of the inference attack against OPE. To reduce the information leakage in OPE, frequency-smoothing solutions [23, 32] were proposed for providing encryption schemes with a security guarantee weaker than the frequency hiding. However, an encrypted database should achieve the stronger security.

## 2.2 Typical FH-OPE Solution

The option of FH-OPE is to generate indeterministic ciphertexts for same plaintexts while preserve orders of ciphertexts. We review existing FH-OPE solutions according to the state.

**Client-state solution**. Kerschbaum [20] proposed the first FH-OPE scheme which stores all the plaintexts in the tree-structure client state. The scheme encrypts same plaintexts into different ciphertexts by introducing random coins which randomizes the position of each plaintext in the tree. The ciphertexts are generated as the mean value of the ciphertexts for the next smaller plaintext and the next greater plaintext. The tree also depicts a mapping from plaintexts to ciphertexts, so that the client can correctly perform decryption operations. However, the client storage is in $O(n)$, which is not practical for large database applications. It motivates us to design a FH-OPE scheme with the small client storage.

**Server-state solution**. The mOPE [30] scheme is an OPE scheme which maintains all ciphertexts in the tree-structure server state. After inserting the ciphertext into the tree, the server will assign an order-preserving encoding to the ciphertext. It is intuitively used to construct the FH-OPE solution by encrypting plaintexts into indeterministic ciphertexts. However, it costs much communication when inserting, because the client must compare the inserted ciphertext with ciphertexts on branch node to decide its correct position on the tree. Moreover, since the server generates the order-preserving encoding of a ciphertext according to its path in the tree, a large number of updates on encodings will be triggered when the tree needs to be rebalanced. To reduce the additional interactions during inserting, Roche et al. proposed POPE [33] that achieves frequency hiding. It sets a B+ buffer tree as the server state, which partially preserves the order of each ciphertext. When the client performs encryption operations, the server only inserts the ciphertext into the buffer of a node. Once the client performs a query operation, ciphertexts in the buffers are evicted to leaf nodes of the tree in order. Although it reduces interactions during inserting, it is still not feasible for encrypted database, since 1) ciphertexts are buffered and not completely ordered, which creates major logic changes on existed encrypted databases; 2) each query operation requires massive additional interactions. These defects motivate us to design a FH-OPE scheme without additional interactions while achieving the low frequency of encoding updates.

# 3 PRELIMINARIES

## 3.1 Stateful Order-preserving Encryption

Considering the ideal security goal (IND-OCPA) for an OPE scheme is infeasible for an encryption model without a state for organizing mutable OPE ciphertexts, our FH-OPE scheme is based on a construction of the stateful OPE scheme. The OPE ciphertext of a plaintext in our scheme is a tuple that consists of 1) a *permanent ciphertext* is encrypted from the plaintext and can be decrypted to recover the plaintext; 2) a *transient order-preserving encoding* indicates the order information of the plaintext.

A stateful OPE scheme OPE = (KeyGen, Setup, Enc, Dec, Order) consists of the following five algorithms.

- $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$: The client generates a secret key sk by the security parameter $\lambda$.
- $\mathsf{st} \leftarrow \mathsf{Setup}(1^\lambda)$: The server initializes the state st by the security parameter $\lambda$.
- $ct, \mathsf{st}' \leftarrow \mathsf{Enc}(\mathsf{sk}, \mathsf{st}, pt)$: The encryption is an interactive algorithm between the client and the server. The inputs to the client are sk and a plaintext $pt$, and the input to the server is the state st. Then, the client obtains a permanent ciphertext $ct$ and the server updates the state from st to st′.
- $pt \leftarrow \mathsf{Dec}(\mathsf{sk}, ct)$: This decryption algorithm is run by the client on sk and a ciphertext $ct$ to obtain a plaintext $pt$.
- $cd \leftarrow \mathsf{Order}(\mathsf{st}, ct)$: This algorithm runs at the server, takes as input a state st and a ciphertext $ct$, and retrieves the order-preserving encoding $cd$.

In OPE, the state st is constructed as a tree stored on the server. This tree is a search tree that maintains a set of *OPE ciphertexts* each of which consists of a *ciphertext* and its *order-preserving encoding*. Take a binary search tree as an example. If there is a node *node* stores the ciphertext $ct$ encrypted from a plaintext $pt$, all the nodes in the left sub-tree of *node* are smaller than $pt$ and all the nodes in the right sub-tree of *node* are larger than $pt$. Moreover, the order-preserving encoding $cd$ of $pt$ is generated based on the position of $ct$ in the tree, so that it can directly indicate the order of $pt$.

**Correctness**. OPE is correct if $\mathsf{Dec}(\mathsf{sk}, ct) = pt$ for any valid state st, $pt$, and $ct, \mathsf{st}' = \mathsf{Enc}(\mathsf{sk}, \mathsf{st}, pt)$. OPE is order-preserving if $cd_i > cd_j \iff pt_i > pt_j$ for any $i$ and $j$.

**Threat model**. In the system of the stateful OPE scheme, there are two entities: the client and the server. The client takes its data as input and wishes to protect the data privacy, while the server is seen as an *honest-but-curious* internal adversary who can loyally perform the OPE algorithms but tries to reveal the data content. The server can know the client's access pattern [9], including which ciphertext is inserted or queried. There is also an outer adversary who plays a role of the eavesdropper. It is as powerful as the honest-but-curious server if it eavesdrops the communication between the client and the server throughout. Therefore, the adversary in Section 3.2 can refer to either this eavesdropper or the honest-but-curious server.

## 3.2 Security Definition

An OPE scheme with IND-OCPA reveals no additional information about the plaintext values besides their orders. Kerschbaum [20] proposed a new security definition for frequency hiding in OPE. It is named indistinguishability under frequency-analyzing ordered

chosen-plaintext attack (IND-FAOCPA) and strictly stronger than IND-OCPA security.

**Randomized order**. In a stateful OPE scheme, a randomized order $\Gamma = \gamma_1, ..., \gamma_n$ of a sequence $PT = pt_1, ..., pt_n$ holds that 1) $1 \leq \gamma_i \leq n$ for any $i$; 2) $i \neq j \Longrightarrow \gamma_i \neq \gamma_j$ for any $i$ and $j$; 3) $pt_i > pt_j \Longrightarrow \gamma_i > \gamma_j$ for any $i$ and $j$; 4) $\gamma_i > \gamma_j \Longrightarrow pt_i \geq pt_j$ for any $i$ and $j$.

**IND-FAOCPA security game**. The security game $Game^{FAOCPA}_{\mathcal{A},\Pi}(\lambda)$ between a challenger and an adversary $\mathcal{A}$ for an OPE scheme $\Pi$ with the security parameter $\lambda$ proceeds as follows.

(1) The adversary $\mathcal{A}$ generates two sequences $PT_0$ and $PT_1$ which have at least one common randomized order $\Gamma$. Each sequence contains $n$ not necessarily distinct plaintexts. Then, $\mathcal{A}$ sends $PT_0$ and $PT_1$ to the challenger.
(2) The challenger generates a secret key $\mathsf{sk} \leftarrow \Pi.\mathsf{KeyGen}(1^\lambda)$ and chooses a random bit $b \in \{0, 1\}$.
(3) The challenger and a server jointly run $\Pi.\mathsf{Setup}(1^\lambda)$ to initialize the state at the server.
(4) Then, the challenger interacts with the server to encrypt $PT_b$ as a ciphertext sequence $CT$. Finally, the challenger sends $CT$ and the state to the adversary $\mathcal{A}$, so that $\mathcal{A}$ can know the access pattern that corresponds to $PT_b$.
(5) The adversary $\mathcal{A}$ outputs a guess $b'$ of $b$.

The adversary $\mathcal{A}$ wins the game if $b' = b$. IND-FAOCPA is defined by the probability of $\mathcal{A}$'s win.

**Definition 1.** *An OPE scheme achieves IND-FAOCPA secure if for all p.p.t. adversaries $\mathcal{A}$, for all $\lambda$, the advantage of outputting $b$ is negligible:*

$$Pr[Game^{FAOCPA}_{\mathcal{A},\Pi}(\lambda) = b] < \frac{1}{2} + \frac{1}{poly(\lambda)}. \quad (1)$$

An OPE scheme with IND-OCPA is also a FH-OPE scheme if it is IND-FAOCPA secure.

## 4 DESIGN GOALS

To meet the frequency-hiding security requirement described in Section 3.2, a stateful OPE scheme should generate and store different OPE ciphertexts for same plaintexts. In this option, the client must encrypt all plaintexts rather than only distinct plaintexts into ciphertexts and then submit the ciphertexts to the server. This makes us fall into a dilemma: we have to cost either a large client storage like the client-state scheme or a number of client-server interactions like the server-state scheme. Alternatively, it inspires us to propose a client-server-state solution which avoids the defects above.

**Local table**. To run with a small client state, we set a *local table* as the state, which maps sorted plaintexts to their counts. Compared with previous client-state schemes, the table reduces the client storage from $O(n)$ to $O(N)$, where $n$ is the total number of plaintexts and $N$ is the number of distinct plaintexts. Moreover, this table requires that the server stores the ciphertexts and organizes them by a server state, which cannot be achieved by previous client-state schemes (e.g., Kerschbaum's scheme). Whenever a new plaintext $pt$ is encrypted, the client carries out how many existing plaintext values are less than $pt$ with the help of the local table. Thus, it is easy to determine a randomized sequential position $pos$ of the corresponding ciphertext $ct$. We hope that the server can insert and further query the ciphertext on $pos$ directly. However, the previous server-state FH-OPE solutions need either extra communication or extra computation to locate $pos$ on the server state even if intuitively adopting the local table. Therefore, we should consider how to build an appropriate server state with this small client storage and further design algorithms of our FH-OPE scheme.

Our FH-OPE solution is supposed to be practically applied in an encrypted database, i.e., like BCLO [4] directly applied in CryptDB [31]. Hence, we have the following two design goals:

- **Without additional interactions**. Similar to the usage of a real-world database, when performing a SQL query, the client only needs to submit a request containing the encrypted SQL query to the server. After the server receives the request, it calls the user-defined function to complete the relevant operation. We require that there is no additional client-server interaction except submitting the request.
- **Reducing frequency of ciphertext re-encoding**. Besides ensuring the IND-OCPA security of an OPE scheme, using the (tree-structure) server state is to maintain each stored ciphertext in order. When this tree needs to be rebalanced, the order-preserving encodings of a set of ciphertexts will be updated. If the tree contains a large number of ciphertexts, this recoding caused by rebalancing will seriously degrade the performance of the OPE scheme. Note that this situation is more common in FH-OPE schemes. For the server-side tree, we require that order-preserving encodings are not updated when the tree is adjusted, so that the frequency of ciphertext re-encoding is reduced.

Then, we describe the server-side tree and the corresponding algorithms of our scheme which achieves the goals.

## 5 CODING TREE

In this section, we give a server-side tree which can produce encodings with the order-preserving property while ensuring the goals in Section 4. Such a tree is called as *coding tree*. We build the coding tree by designing a coding strategy and constructing the appropriate structure.

### 5.1 Coding Strategy

In the OPE scheme with a server-side tree, the order-preserving encoding indicates the path of its ciphertext stored in this sort tree. The ciphertext's path information intuitively represents the order of its corresponding plaintext. That is the reason why the order-preserving property of such an encoding works. Alternatively, balancing this tree is precisely what mutates the path-related encoding, since the encoding must be updated according to the change of its path. Hence, it is difficult to reduce the frequency of recoding this encoding. This path-related encoding is quite double-blade.

To avoid the inherent defect of path-related encodings, we give a *region-code strategy* to encode each ciphertext into an order-preserving encoding which shows the order of its underlying plaintext. Instead of recoding during the rebalancing in previous works, our coding strategy updates encodings only if necessary.

We set that each leaf node in the coding tree stores some ciphertexts within a specific interval. The interval of a node is denoted as

$(lower, upper]$. When inserting a new ciphertext into this node, the server takes the middle value of two neighbours' encodings as the ciphertext's encoding. If the left (right) neighbour is empty, we use the lower (upper) bound instead. For example, if a new ciphertext is decided to insert in the position between $cd_1$ and $cd_2$, its encoding is set as $\lceil (cd_1 + cd_2)/2 \rceil$. These encodings can be utilized to make an order comparison between any two ciphertexts. Although tree rebalancing operations change some nodes' paths, order relationships of all nodes are still preserved. Therefore, there is no need for updating our encodings which are only related to the orders. We will show this property in Section 5.3.2.

When our coding strategy works, the condition that triggers the recoding is that there is no available encoding for the new ciphertext. In this situation, a set of adjacent ciphertexts in the same node or brother nodes will be encoded again in a uniform way. For example, there is a node with an interval $(0, 8]$. It contains a ciphertexts sequence $\{ct_1, ct_2, ct_3\}$ with the corresponding encodings are $\{1, 3, 4\}$. When inserting a new ciphertext $ct'$ between $ct_2$ and $ct_3$, the encodings in the node will be extended to $\{2, 4, 6, 8\}$ which indicate orders of the sequence $\{ct_1, ct_2, ct', ct_3\}$. This situation is rare, which is demonstrated by the evaluation in section 8.4.

## 5.2 Structure of Coding Tree

To enable the coding strategy above, the coding tree should support not only searching from its root but also the sequential traversal in left-to-right order. Therefore, we choose a new structure to construct our coding tree.

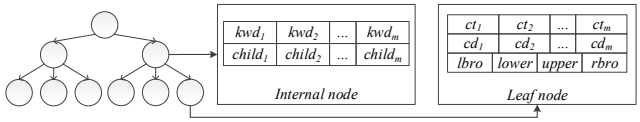



**Figure 1: Node Component of Coding Tree.**

**Node**. We take the $m$-order coding tree as an example. Each internal node has at most $m$ child nodes and each leaf node stores at most $m$ ciphertexts. We denote $imax$ as the current number of elements on a node. Each node's component is shown as follows.

- **Internal node**. An internal node contains the pointer $child$ and the corresponding keyword $kwd$ for each child node. These keywords can help to finish insert operations and traversal operations.
- **Leaf node**. A leaf node stores each inserted ciphertext $ct$ with its transient encoding $cd$. In addition, the leaf node also contains its encoding interval $(lower, upper]$ and the point of its brother nodes. When a node has more than $m$ ciphertexts, it will split into two nodes which are brothers. Such a rebalancing does not cause any recoding operations.

Figure 1 depicts an internal node with $imax$ child nodes and a leaf node stores $imax$ ciphertexts.

**Keyword**. It is supposed that only given a sequential position $pos$, the server can make use of keywords of internal nodes to locate $pos$ from the root. The coding tree has a structure similar to the B+ tree, but we observe that the keywords of the B+ tree and the corresponding search way are not suitable for our location operations. Hence, we define that each keyword in an internal node represents the number of OPE ciphertexts contained in the descendants of the corresponding child node.

**Usage of keyword**. When receiving the request that contains a ciphertext $ct$ with its position $pos$, the server inserts $ct$ from the root of the coding tree. Let $kwd_i$ be the $i$-th keyword of the current node and the tree be $m$-order, the server:

1 for each $i = \{1, 2, ...\}$, when $pos > kwd_i$ which means that ciphertext $ct$ will not be inserted into the descendants of $child_i$, repeat $pos \leftarrow pos - kwd_i$ until $pos \leq kwd_i$;
2 set $child_i$ as the current node and repeat step 1 until the current node is a leaf node;
3 insert $ct$ into the current node as the $(pos + 1)$-th ciphertext;
4 if there is no available encoding for $ct$, perform a recoding operation, and otherwise give it an encoding;
5 if the node stores $m + 1$ ciphertexts now, perform a tree rebalancing operation.

## 5.3 Building Blocks of Coding Tree

To process basic operations of our coding tree, we define some functions — building blocks run by the server. Figure 2 describes the building blocks in detail.

Insert$(root, pos, ct)$. This function takes the coding tree $root$, a target ciphertext $ct$, and its position $pos$ as input. The server makes a comparison between $pos$ and each keyword of the current node to recursively locate the leaf $leaf$ which $pos$ belongs to. Then, the server calls Encode$(leaf, pos + 1)$ to get the encoding of $ct$ and the range of updated encodings. If $leaf$ stores $m + 1$ ciphertexts after inserting, the server performs the tree rebalancing Rebalance$(leaf)$ for $leaf$. The function returns $ct$'s encoding. If Encode involves the update on encodings, the function also returns the range $(lower, upper]$ of updated encodings.

Encode$(leaf, pos)$. This function takes a leaf $leaf$ and a target position $pos$ of it as input. Assume that the ciphertext $ct$ is in $pos$ of $leaf$. The server encodes $ct$ as $cd$ which is the middle value of encodings of $ct$'s two neighbours. Then, if there is no available encoding for $ct$, the server calls the building block Recode$(leaf)$ to recode each ciphertext on $leaf$. The function returns $cd$. If there is any recoding, the function also returns the recoding range $(lower, upper]$.

Recode$(leaf)$. This function takes a leaf $leaf$ as input. It reallocates encodings for ciphertexts on $leaf$ or its brothers when triggering the recoding operation during encoding. We denote $imax$ as the number of elements on a node. If there are more than $leaf.imax$ integers on the interval of $leaf$, the server uniformly allocates them for the ciphertexts. Otherwise, $leaf$'s brother nodes will be gradually aggregated as a block-linked list for encoding operations. In particular, when the leftmost/rightmost leaf does not have enough interval to allocate encodings, the server will double the leaf's lower/upper bound. Finally, the function outputs the recoding range $(lleaf.lower, rleaf.upper]$ at the leaf level. Note that the recoding is easy to implement, since the coding tree supports sequential searches at the leaf level. Figure 3 shows an example of the recoding. When inserting (Figure 3(a)) a new ciphertext between $ct_1$ and $ct_2$ on the first leaf, the tree reallocate the encodings on this leaf since there is no available encoding between 0 and 1 (Figure

Encode($leaf, pos$)
1: $left, right \leftarrow leaf.lower, leaf.upper$;
2: $lower, upper \leftarrow \perp, \perp$;
2: **if** $pos > 1$
3: $left \leftarrow leaf.cd_{pos-1}$;
5: **if** $pos < leaf.imax$
6: $right \leftarrow leaf.cd_{pos}$;
7: $cd \leftarrow \lceil \frac{left+right}{2} \rceil$;
8: **if** $|right - left| \leq 1$
9: $lower, upper \leftarrow$ Recode($leaf$);
10: $cd \leftarrow leaf.cd_{pos}$;
11: **return** $cd, lower, upper$.

Recode($leaf$)
1: $lleaf, rleaf \leftarrow leaf, leaf$;
2: $imax \leftarrow leaf.imax$;
3: **if** $(rleaf.upper - lleaf.lower) \geq imax$
4: $frag \leftarrow \lfloor \frac{rleaf.upper - lleaf.lower}{imax} \rfloor$;
5: $cleaf \leftarrow lleaf$;
6: $cd \leftarrow lleaf.lower$;
7: **for** each $i \in [cleaf.imax]$
8: $cd \leftarrow cd + frag$;
9: $cleaf.cd_i \leftarrow cd$;
10: **if** $cleaf \neq rleaf$
11: $cleaf.upper \leftarrow cd$;
12: $cleaf \leftarrow cleaf.rbro$;
13: $cleaf.lower \leftarrow cd$;
14: **goto** 7;
15: **else**
16: **if** $lleaf.lbro \neq NULL$
17: $lleaf \leftarrow lleaf.lbro$;
18: $imax \leftarrow imax + lleaf.imax$;
19: **else if** $lleaf.lower < 0$
20: $lleaf.lower \leftarrow lleaf.lower \cdot 2$;
21: **if** $rleaf.rbro \neq NULL$
22: $rleaf \leftarrow rleaf.rbro$;
23: $imax \leftarrow imax + rleaf.imax$;
24: **else**
25: $rleaf.upper \leftarrow rleaf.upper \cdot 2$;
26: **goto** 3;
27: **return** $lleaf.lower, rleaf.upper$.

GetCode($node, pos$)
1: **if** $node$ is leaf
2: $cd \leftarrow node.cd_{pos}$;
3: **else**
4: **for** $i \in [1, node.imax]$;
5: **if** $pos > node.kwd_i$
6: $pos \leftarrow pos - node.kwd_i$;
8: **else**
8: **break**;
9: $cd \leftarrow$ Getcode($node.child_i, pos$);
10: **return** $cd$.

Insert($node, pos, ct$)
1: **if** $node$ is leaf
2: Insert $ct$ into $node$ as the $(pos+1)$-th ciphertext;
3: $node.cd_{pos+1}, lower, upper \leftarrow$ **Encode**($node, pos+1$);
4: **if** $node.imax > m$
5: Rebalance($node$);
6: **return** $node.cd_{pos+1}, lower, upper$;
7: **else**
8: **for** $i \in [1, node.imax]$;
9: **if** $pos > node.kwd_i$
10: $pos \leftarrow pos - node.kwd_i$;
11: **else**
12: **return** Insert($node.child_i, pos, ct$).

Rebalance($node$)
1: **if** $node$ is leaf
2: Create a new leaf $node'$;
3: Move $node$'s last $\lfloor \frac{m}{2} \rfloor$ ciphertexts with their encodings to $node'$;
4: **else**
5: Create a new internal node $node'$
6: Move $node$'s last $\lfloor \frac{m}{2} \rfloor$ children with their keywords to $node'$;
7: **if** $node.parent = NULL$
8: Create a new internal node $root$ as the new root of the coding tree;
9: Insert $node$ into $root$;
10: Insert $node'$ into $node.parent$ as $node$'s right brother;
11: **if** $node.parent.imax > m$
12: Rebalance($node.parent$);
13: **return**.

**Figure 2: Building Blocks on Server Side**

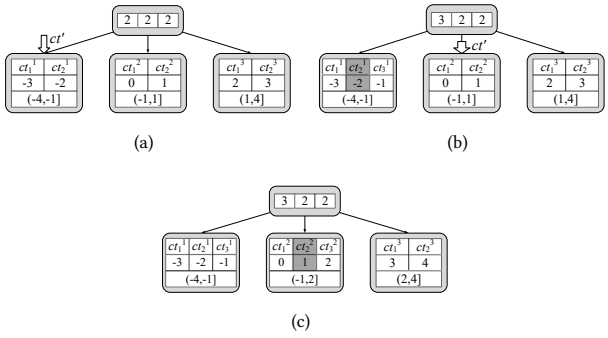



**Figure 3: An Example for Recoding**

3(b)). Then, the recoding occurs on both the second leaf and the third leaf when inserting another new ciphertext (Figure 3(c)).

Rebalance($node$). This function takes a node $node$ as input. If $node$ stores more that $m$ elements, i.e., $node.imax > m$, it rebalances the coding tree from $node$. For the coding tree, the function recursively splits the overloaded nodes in the tree. When $node$ needs to be split into two nodes, its last $\lfloor \frac{m}{2} \rfloor$ elements are moved to a new node $node'$. Then, $node'$ is inserted into $node$'s parent as $node$'s right brother. In particular, if the node that will be split is the root of the tree, the server creates a new root as the node's parent.

GetCode($root, pos$). This function takes the coding tree $root$ and a target position $pos$ as input, and returns the $pos$-th encoding $cd$ of the tree.

*5.3.1 Usage of Local Table.* Then, we show how to generate the OPE ciphertext for a plaintext $pt$ by using our coding tree with our client local table.

**Ciphertext**. As mentioned in Section 3.2, a FH-OPE scheme must have a tie-breaking mechanism for hiding the frequency of

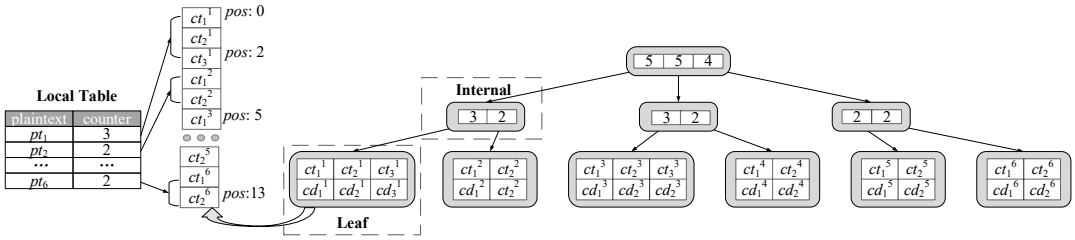



**Figure 4: A Example of OPE Ciphertext Generation using Local Table**

each plaintext in permanent ciphertexts. For generating indeterministic permanent ciphertexts, we will adopt a standard IND-CPA symmetric key encryption scheme RND = (KeyGen, Enc, Dec) to encrypt plaintexts. Thus, the client can encrypt $pt$ into $ct \leftarrow \mathsf{RND.Enc}_k(pt)$, where $k$ is the symmetric encryption key.

The ciphertexts encrypted from $pt$ are supposed to be allocated different tie-breaking encodings while still keeping order-preserving in the coding tree. After the client encrypts $pt$ into $ct$, we set that the indeterministic position $pos$ is carried out by uniformly sampling an integer from $\{\sum_{pt'<pt} T[pt'], ..., \sum_{pt'\leq pt} T[pt']\}$. Then, the client updates the local table as follows: 1) if $pt$ has already existed in the table, add 1 to its count; 2) otherwise, insert $pt$ into the table and set its count as 1. Finally, the client submits $ct$ along with $pos$ to the server. The server calls Insert to insert $ct$ into the coding tree and gives it a transient order-preserving encoding $cd$. The OPE ciphertext $\{ct, cd\}$ is stored on the server. Figure 4 depicts the coding tree with the local table and the condition after encrypting a new plaintext $pt'$ using the local table.

*5.3.2 The Example of Rebalancing.* We present a simple example to explain how the coding tree unbinds the tree rebalancing and the recoding. In another word, it can show the relationship between Rebalance and Recode when inserting random plaintexts.

We consider the encryption on a binary plaintext domain of "gender". There are two possible plaintext values: "male" (0) and "female" (1). We build a 3-order coding tree and set the interval of the root as $(-8, 8]$. We have 8 plaintexts $\{0, 1, 0, 1, 0, 1, 0, 1\}$ in all and we have already inserted following sequence $\{0, 1, 0, 1, 0, 1, 0\}$ into the coding tree. The current states — the local table and the coding tree are depicted in Figure 5(a), where $E(\cdot)$ is the shorthand of the encryption method for generating ciphertexts.

We show the operation where the client encrypts plaintext 1. The client carries out a random position $pos = 6$ and submits it along with $E(1)$ to the server. The server inserts $E(1)$ into the third leaf and assigns it an encoding 5. The situation is depicted in Figure 5(b) and triggers Rebalance since the number of ciphertexts stored on this leaf is larger than $m$. The building block Rebalance is called twice recursively. The situation after rebalancing is shown in Figure 5(c) and we can see that any encoding is not updated.

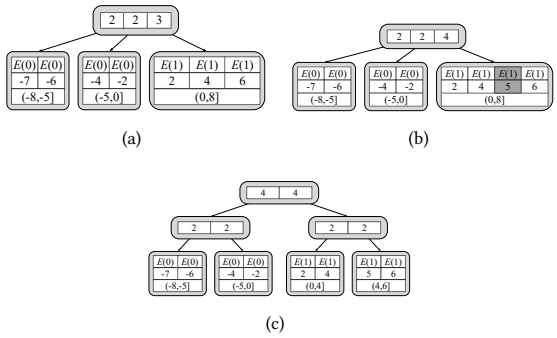



**Figure 5: An Example for Inserting Ciphertexts**

## 6 ALGORITHMS

In this section, we present the algorithms of our FH-OPE scheme. Extended from the definition of stateful OPE, our FH-OPE scheme FH-OPE = (KeyGen, Setup, Enc, Dec, Search) consists of the following five algorithms.

- $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$: The client generates a secret key sk by the security parameter $\lambda$.
- $\mathsf{st_{Cl}}, \mathsf{st_{Ser}} \leftarrow \mathsf{Setup}(1^\lambda)$: By the security parameter $\lambda$, the server initializes the state $\mathsf{st_{Ser}}$ that contains an empty coding tree, while the client initializes the local state $\mathsf{st_{Cl}}$ that contains an empty table.
- $ct, \mathsf{st'_{Cl}}, \mathsf{st'_{Ser}} \leftarrow \mathsf{Enc}(\mathsf{sk}, \mathsf{st_{Cl}}, \mathsf{st_{Ser}}, pt)$: The encryption is an algorithm involved both the client and the server. Given an input, the client obtains a permanent ciphertext $ct$ and updates the local table in $\mathsf{st_{Cl}}$, while the server updates the coding tree in $\mathsf{st_{Ser}}$.
- $pt \leftarrow \mathsf{Dec}(\mathsf{sk}, ct)$: This decryption algorithm is run by the client on sk and a ciphertext $ct$ to obtain a plaintext $pt$.
- $cd \leftarrow \mathsf{Search}(\mathsf{st_{Cl}}, \mathsf{st_{Ser}}, pt)$: The search is an algorithm involved both the client and the server. The inputs to the client are the local table and a plaintext $pt$. Then, the server retrieves the encoding $cd$ of $pt$.

**KeyGen**. As shown in Algorithm 1, the client generates a secret key sk by running the encryption scheme of RND and then returns sk. The key sk is used for encryption and decryption operations.

**Algorithm 1** $\mathsf{KeyGen}(1^\lambda)$

**Require:**
the security parameter $\lambda$
**Ensure:**
the secret key sk
1: $\mathsf{sk} \leftarrow \mathsf{RND.KeyGen}(1^\lambda)$;
2: **return** sk.

**Algorithm 2** Setup($1^\lambda$)

**Require:**
**Server/Client**: the security parameter $\lambda$;
**Ensure:**
**Client**: the local state $\mathsf{st_{Cl}}$;
**Server**: the state $\mathsf{st_{Ser}}$
**Client**:
1: Initialize an empty table $T \leftarrow \emptyset$;
2: Set $T$ as $\mathsf{st_{Cl}}$;
**Server**:
3: Initialize a leaf node $root$ as tree root;
4: $root.lower, root.upper \leftarrow default_l, default_r$;
5: **return** $\mathsf{st_{Cl}}$ and $\mathsf{st_{Ser}}$.

**Setup**. The Setup algorithm is shown in Algorithm 2. The client creates an empty local table $T$ for recording the count of each plaintext. Alternatively, the server builds a $m$-order coding tree contains an empty leaf node $root$ as root. The interval of the root is set as $(default_l, default_r]$ where $default_l$ is the default lower bound for order-preserving encoding and $default_r$ is the default upper bound for order-preserving encoding, $(default_l, default_r) \in \mathbb{Z}$. The table $T$ and the tree $root$ as initial states for the client and the server, respectively.

**Algorithm 3** Enc($\mathsf{sk}, \mathsf{st_{Cl}}, \mathsf{st_{Ser}}, pt$)

**Require:**
**Client**: the secret key sk, the local state $\mathsf{st_{Cl}}$, and a plaintext $pt$;
**Server**: the state $\mathsf{st_{Ser}}$
**Ensure:**
**Client**: the updated local state $\mathsf{st'_{Cl}}$ and the ciphertext $ct$;
**Server**: the updated state $\mathsf{st'_{Ser}}$
**Client**:
1: $ct \leftarrow \mathsf{RND.Enc}(\mathsf{sk}, pt)$;
2: Get $T$ from $\mathsf{st_{Cl}}$;
3: $l \leftarrow \sum_{pt'<pt} T[pt']$;
4: **if** $pt \in T$ **then**
5: Uniformly sample $pos \xleftarrow{R} \{l, l+1, ..., l+T[pt]\}$;
6: $T[pt] \leftarrow T[pt] + 1$;
7: **else**
8: $pos \leftarrow l$;
9: Insert $pt$ into $T$;
10: $T[pt] \leftarrow 1$;
11: **end if**
12: Set $T$ as $\mathsf{st'_{Cl}}$;
13: Send $pos$ and $ct$ to **Server**;
**Server**:
14: Get $root$ from $\mathsf{st_{Ser}}$;
15: Call Insert($root, pos, ct$);
16: Set the coding tree $root$ as $\mathsf{st'_{Ser}}$;
17: **return** $ct$, $\mathsf{st'_{Cl}}$, and $\mathsf{st'_{Ser}}$.

**Encryption**. The Enc algorithm is shown in Algorithm 3. For an input plaintext $pt$, the client uses the in-determinant scheme RND to encrypt it as a ciphertext $ct$. Then, the client can count the number of existing plaintext values smaller than $pt$ by computing $l \leftarrow \sum_{pt'<pt} T[pt']$. Note that at the leaf level of the coding tree, $l$ is the lower bound of the position $pos$ where $ct$ will be located. Therefore, the positions of ciphertexts encrypted from $pt$ range from $l+1$ to $l+T[pt]$ and $ct$ is inserted after one of them. After updating $T$, the client uploads $ct$ along with $pos$ to the server. Finally, the server runs the building block Insert($root, pos, ct$) to insert $ct$ into the tree and give it a transient OPE encoding.

**Algorithm 4** Dec($\mathsf{sk}, ct$)

**Require:**
the secret key sk and a ciphertext $ct$
**Ensure:**
the plaintext $pt$
1: $pt \leftarrow \mathsf{RND.Dec}(\mathsf{sk}, ct)$;
2: **return** $pt$.

**Decryption**. The algorithm (Algorithm 4) takes private key sk, ciphertext $ct$ as input and outputs the corresponding plaintext $pt$.

**Algorithm 5** Search($\mathsf{st_{Cl}}, \mathsf{st_{Ser}}, pt$)

**Require:**
**Client**: the local state $\mathsf{st_{Cl}}$ and a plaintext $pt$;
**Server**: the state $\mathsf{st_{Ser}}$
**Ensure:**
**Server**: the transient OPE encoding $cd$
**Client**:
1: Get $T$ from $\mathsf{st_{Cl}}$;
2: $pos \leftarrow \sum_{pt'<pt} T[pt'] + 1$;
3: Send $pos$ to **Server**;
**Server**:
4: Get $root$ from $\mathsf{st_{Ser}}$;
5: $cd \leftarrow \mathsf{GetCode}(root, pos)$;
6: **return** $cd$.

**Search**. The Search algorithm is shown in Algorithm 5. If the client wants to search the minimal encoding of $pt$, it will run this algorithm with the server. Take the SQL application as an example. If the client queries a SQL statement like "select * from table where $ID >= r_{min}$ and $ID < r_{max}$" to server, the SQL statement will be rewritten as a secure form: "select * from table where $ID >=$ $\mathsf{Search}(\mathsf{st_{Cl}}, \mathsf{st_{Ser}}, r_{min})$ and $ID <$ $\mathsf{Search}(\mathsf{st_{Cl}}, \mathsf{st_{Ser}}, r_{max})$".

## 6.1 Security Analysis

We give a proof of IND-FAOCPA security of our scheme in Section 6. The definition says that an adversary cannot distinguish between encryptions of two challenge sequences as long as the sequences have at least one common randomized order. We prove by constructing two hybrid games in which the adversary $\mathcal{A}$ produces identical outputs for anything it views. We denote our scheme as $\Pi$ and define the two hybrids as follows.

- **Hybrid 1**. It is the IND-FAOCPA game (see Section 3.2) whose output is $Game^{FAOCPA}_{\mathcal{A},\Pi}(\lambda)$.
- **Hybrid 2**. We change the IND-FAOCPA game to **Hybrid 2** by modifying $\Pi$.Enc. Before encryption, the challenger uses the (polynomial-time) Algorithm 6 in [20] to randomly

select a common randomized order $\Gamma = \{\gamma_1, ..., \gamma_n\}$ to $PT_0$ and $PT_1$ from the set of all common randomized orders. Assume that $\gamma_i$ is the randomized order of the $i$-th plaintext $pt_i^b$ in $PT_b$. During encryption, as locating ciphertexts in the server-side state $\mathsf{st}_{\mathsf{Server}}$ (the coding tree), the challenger uses the randomized order values. That is, the challenger sets $pos \leftarrow \sum_{j<i} count(\gamma_j < \gamma_i)$ instead of uniformly sampling from $\{l, l+1, ..., l+T[pt]\}$. Then, it sends $ct_i$ along with $pos$ to the server.

In the following lemma, we prove that the chance that $\mathcal{A}$ guesses correctly in **Hybrid 2**, i.e., the overall winning advantage of the adversary can at most be $\frac{1}{2} + negl(\lambda)$.

Lemma 1. *For any p.p.t. adversary $\mathcal{A}$, the chance that $\mathcal{A}$ guesses correctly in **Hybrid 2** is $\frac{1}{2} + negl(\lambda)$.*

Proof. We consider any adversary $\mathcal{A}$ and any two plaintext sequences of values $\mathcal{A}$ asks for in **Hybrid 2**: $PT_0 = \{pt_1^0, ..., pt_n^0\}$ and $PT_1 = \{pt_1^1, ..., pt_n^1\}$. The view of $\mathcal{A}$ consists of a ciphertext sequence $CT$ and the server-side state $\mathsf{st}_{\mathsf{Ser}}$ receives in **Hybrid 2**. We consider two cases: the case when the challenger chose $PT_0$ to encrypt and the case when the challenger chose $PT_1$ to encrypt. We argue that the view of $\mathcal{A}$ in the two cases is exactly the same.

When no value was encrypted and we can see that $\mathcal{A}$ starts off with the same information. According to the algorithm $\Pi$.Enc, given a plaintext $pt_i^b$, the challenger will encrypt and upload it whether the server has already stored an encryption of it. Hence, the ciphertext sequence $CT = \{CT_1, ..., CT_n\}$ must be $n$-length. By observing the encryption of the $i$-th plaintext $pt_i^b$, $\mathcal{A}$ can only guess $b$ correctly with negligible probability in both cases due to the security of RND. This situation for each $i \in [n]$ also remains the same.

Alternatively, the position $pos$ of the ciphertext $ct_i$ in **Hybrid 2** is deterministic, since the choice of $\gamma_i$ is determined by the common randomized order $\Gamma$ uniformly selected among all possible randomized orders. By observing the encoding of the $i$-th plaintext from $\mathsf{st}_{\mathsf{Ser}}$, $\mathcal{A}$ obtains the same information distribution in both cases and the information for each $i \in [n]$ also remains the same.

Therefore, in both cases $\mathcal{A}$ receives the same information and cannot distinguish with non-negligible probability. □

**Theorem 1.** *Our scheme is IND-FAOCPA secure.*

Proof. It is straightforward to check that the two hybrids can produce a same output for the adversary $\mathcal{A}$. Hence, the probability that the adversary wins **Hybrid 1** against our scheme is negligible in $\lambda$, which meet the definition of IND-FAOCPA in Section 3.2. □

## 7 USAGE IN ENCRYPTED DATABASE

**Our scheme in encrypted database**. There is a way to implement our FH-OPE solution in the encrypted database without rewriting database drivers. Similar to description of mOPE [30], we implement and encapsulate our scheme in UDFs of MySQL. The implementation includes a client application and a server database. The client stores the local table $T$ and submits requests when performing inserts and queries. The server uses the UDF to finish operations on the database using the coding tree *root* and a small table $M$ mapping from ciphertexts to encodings. The server runs the block Setup to initialize the coding tree and sets $M$ as an empty table. In this implementation, our local table strictly keeps the consistency of client data and server data, which avoids the error caused by wrong encodings of ciphertexts. These UDFs are shown as follows.

- *FHOPE_Insert*$(pos, ct)$. It takes a ciphertext $ct$ and its position $pos$ as input. Then, it calls the block $Insert(root, pos, ct)$ to insert the $ct$ into $pos$ on the coding tree. If the insert operation does not involve any encoding update, the UDF returns $ct$'s encoding $cd = Insert(root, pos, ct)$. Otherwise, the UDF uses $M$ to remember all updated encodings.
- *FHOPE_Update*$(ct)$. It takes a ciphertext $ct$ as input and returns $cd$ which is the corresponding encoding of $ct$ in $M$.
- *FHOPE_Update_Upper*()/*FHOPE_Update_Lower*(). It returns the upper/lower bound of encodings in $M$.
- *FHOPE_Search*$(pos)$. It takes a target position $pos$ as input. Then, it calls the block $cd = GetCode(root, pos)$ to query the ciphertext in the position $pos$ on the coding tree. Finally, the UDF returns the encoding $cd$.

**Example of usage**. We show how to use the UDF to process operations on an encrypted table "*table example (ciphertext varchar(128), encoding bigint)*" as on the corresponding plaintext table "*table example (plaintext varchar(16))*"

- **Insert**. When the client tries to insert a ciphertext $ct$ encrypted from $pt$ into the database, it submits the sql "*call PRO_INSERT(pos, ct)*" instead of "*insert into example values (pt)*". The procedure is defined as follows.
  *create procedure PRO_INSERT (IN pos int, IN ct varchar(128))*
  *BEGIN*
  *insert into example values (ct,FHOPE_Insert(pos, ct));*
  *if FHOPE_Update_Lower() < FHOPE_Update_Upper()*
  *then update example set encoding = FHOPE_Update(ciphertext)*
  *where encoding > FHOPE_Update_Lower() and encoding <= FHOPE_Update_Upper()"; end if;*
  *END*
- **Query**. When the client tries to perform a range query from $pt_{min}$ to $pt_{max}$, it submits the sql "*select * from example where encoding > FHOPE_Search( $pos_{min}$ ) and encoding < FHOPE_Search( $pos_{max}$ )*" where $pos_{min} = \sum_{pt \le pt_{min}} T[pt]$ and $pos_{max} = \sum_{pt \le pt_{max}} T[pt] + 1$, instead of "*select * from example where plaintext > $pt_{min}$ and plaintext <= $pt_{max}$*".

Note that our scheme can be modified to support deletions further. If the client wants to run a sql "*delete from example where encoding > $pt_{min}$ and encoding <= $pt_{max}$*", the client deletes $T[pt]$ in the local table for each $pt$ in range $(pt_{min}, pt_{max}]$ and runs the sql "*delete from student where age > FHOPE_Search( $pos_{min}$) and age < FHOPE_Search( $pos_{max}$)*" instead. Such that the corresponding ciphertexts in the coding tree are deleted.

## 8 EVALUATION

### 8.1 Implementation Details

Our evaluation is conducted by our implementation (shown in Section 7) on MySQL. The source code is shared in GitHub[1]. This implementation includes a *client* application and a *server* database.

[1] https://github.com/LiDongJieHIHA/OPEUDF/

The client application is implemented by Python 3 on a local machine equipped with an Intel(R) Core(TM) i7-6700 GPU @3.40 GHz, 16 GB available RAM, Windows 10 desktop version. The database UDFs is implemented by C++ on a remote server equipped with 8-core Intel(R) Xeon W-2245 GPU @3.9GHz and 64 GB available RAM and installed with Ubuntu 18.04. This server is provided by RDS MySQL of Aliyun with 5Mbps uplink/downlink bandwidth. We use AES in the Pycrypto library with randomness to implement the encryption RND. The encryption key of AES is set to 128 bits. The coding tree in our scheme is set as 128-order. To make our scheme deal with negative encodings correctly, we set a sign bit for each encoding.

## 8.2 Experiment Setting

We evaluate our scheme by comparing the experimental performance between our scheme and other schemes on *client storage*, *recoding frequency*, *interaction*, and *overall application*. The project *overall application* can depict a scheme's overall performance in the real-world database application. We set three schemes as the references for our experiments. We implement Kerschbaum's scheme [20] as a reference of the client-state scheme. We modify mOPE [30] with a frequency-hiding method and regard it as a reference of the server-state scheme. That is, we remain all properties of mOPE except randomly generating different ciphertexts for same plaintexts. We implement POPE [33] as another reference of the server-state scheme, which does not provide order-preserving encodings and only partially preserves the ciphertext's order. Moreover, POPE buffers each encrypted ciphertexts, which means it does not keep the ciphertexts in order until the first query reaches. Therefore, we mainly evaluate its performance on *interaction* and *overall application* by generally comparing it to our scheme. All implementations work in the single-thread mode.

**Dataset**. Our experiments are performed on three datasets including two *synthetic distributions* and a *real dataset*. Two synthetic distributions are uniform (denoted as *Distribution* 1) and normal (denoted as *Distribution* 2), respectively. Given $n$, *Distribution* 1 can be used for constructing a dataset contains $N$ distinct plaintexts by changing the range, where $n$ is the size of the dataset. Similarly, we can use *Distribution* 2 for constructing the dataset by changing the standard deviation. Such that we can generate datasets for evaluating the influence of $N/n$ on the client storage. The real data set is California public employees salaries [3] which contains 247697 records with 6 columns. We choose "Job Title" and "Other Pay" fields of which details are shown in Table 2 for our experiments. To highlight the frequency-hiding property of OPE schemes, we round each float number to an integer. This real dataset is used for evaluating the functionality of our encrypted database UDF.

**Input sequence**. When evaluating the project including insert/query operations, e.g., *recoding frequency* and *interaction*, the input sequence of plaintexts is an important fact for affecting the performance of a scheme. This is because different input sequences will result in different tree conditions which may make the tree-structure state unbalance. In our experiments, we consider three input sequences as follows.

- **The best case** ($-best$). Plaintexts are uniformly set in the input sequences according to their orders.
- **The worst case** ($-worst$). Plaintexts are incrementally/ decrementally set in the sequences according to their orders.
- **The natural case** ($-natural$). Plaintexts are naturally listed in the sequences without manual adjustments. This case is only for the real dataset and most common in applications.

**Table 2: Columns in Real Data Set**

| Field | Distinct Plaintext | Dataset Size | $N/n$ |
|---|---|---|---|
| Job Title | 3826 | 247697 | 0.015 |
| Other Pay | 100598 | 247697 | 0.406 |

## 8.3 Storage

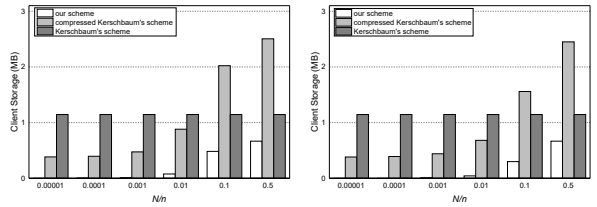
We test the client storage of our scheme and make comparisons between our scheme and client-state schemes. Besides Kerschbaum's scheme[20], we set up another Kerschbaum's scheme with a client state which is compressed by the approach in [20]. We denote it as compressed Kerschbaum's scheme. Note that compressed Kerschbaum's scheme performs poorer than Kerschbaum's scheme in any project except for the client storage, such that we only use it as the reference in this experiment. We evaluate the client storage from two dimensions: the number of plaintexts (i.e., $n$) and the proportion of distinct plaintexts (i.e., $N/n$).

By adjust *Distribution* 1 and *Distribution* 2, we construct several experimental datasets of which $N/n$ varies from 0.00001 to 0.5. Figure 6 depicts these experimental results. Fixing $n$ ($n = 10^5$ or $n = 10^6$), the client storage size of our scheme is approximately in $O(N)$, while that of Kerschbaum's scheme and compressed Kerschbaum's scheme is approximately in $O(n)$. Our client storage size is always smaller. Table 3 shows the results evaluated on two columns (i.e., Job Title and Other Pay) of the real datasets. On the chosen columns, the client storage size of our scheme is smaller than that of other schemes, which is similar to the results on the synthetic distributions.

**Table 3: Client Storage Cost (Mbyte)**

| Dataset Size | our scheme | | K's scheme | | compressed K | |
|---|---|---|---|---|---|---|
| | Job Title | Other Pay | Job Title | Other Pay | Job Title | Other Pay |
| 1000 | 0.0010 | 0.0041 | 0.0152 | 0.0114 | 0.0073 | 0.0164 |
| 10000 | 0.0069 | 0.0436 | 0.1525 | 0.11443 | 0.0616 | 0.18022 |
| 100000 | 0.0302 | 0.1314 | 1.5259 | 1.1444 | 0.4967 | 0.9147 |
| 247697 | 0.0438 | 0.1754 | 3.7795 | 2.8347 | 1.1226 | 1.6921 |

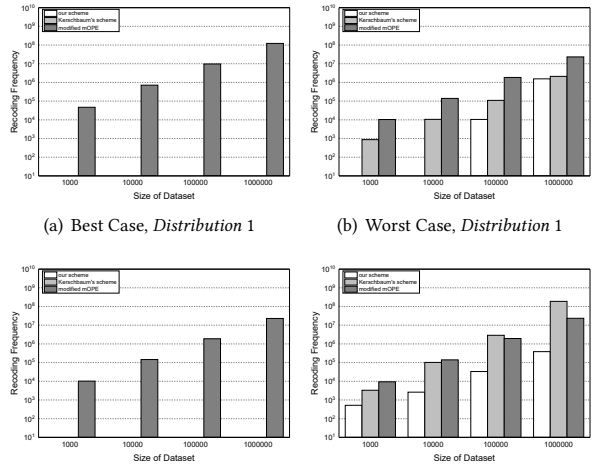
The advantage of our scheme is obvious when $N/n$ is small (i.e., $N << n$). This condition is very common for the dataset of which frequency needs to be hidden. In this condition, the frequency of plaintexts has obvious distribution characteristics which make the OPE scheme easy to suffer inference attacks. Note that our scheme still outperforms schemes with a client state (e.g., Kerschbaum's scheme and compressed Kerschbaum's scheme) even if $n$ and $N$ are close (e.g., $N = 0.5n$), since our local table is in $O(N)$ and other client states are in $O(n)$. Considering there is always $N \leq n$, our scheme will not degrade to (compressed) Kerschbaum's solution.




(a) $n = 10^5$, *Distribution* 1 (b) $n = 10^5$, *Distribution* 2

(c) $n = 10^6$, *Distribution* 1 (d) $n = 10^6$, *Distribution* 2

**Figure 6: Client Storage Cost**




(a) Best Case, *Distribution* 1 (b) Worst Case, *Distribution* 1

(c) Best Case, *Distribution* 2 (d) Worst Case, *Distribution* 2

**Figure 7: Encoding Update Frequency**

**Server-side storage**. For a commercial encrypted database, bottlenecks of the system performance are often communicational overhead and client storage cost, since the server has the powerful capability for storing. Alternatively, the server state (the coding tree in our scheme) has a negligible size compared with all ciphertexts in the whole database. Therefore, our server storage cost is still in $O(n)$, which is acceptable for practical applications.

## 8.4 Encoding Update Frequency

We evaluate the frequency of encoding updates by inserting a number of plaintexts (the client-state scheme) or ciphertexts (the server-state scheme and ours) into the tree-structure state.

Figure 7 depicts the experimental results on two synthetic distributions when the input sequence is in $-best$ case and $-worst$ case. In both cases, the recoding frequency in the modified mOPE scheme is very high and constant, since the recoding is performed whenever a ciphertext is inserted. In $-best$, there is almost no recoding that occurs in our scheme and Kerschbaum's scheme. In $-worst$, the frequency of recoding operations in Kerschbaum's scheme is very high, which is more than 10 times of that in our scheme. Table 4 shows the result on the real dataset, which is similar to the result on synthetic datasets. Note that $-natural$ case of this real dataset is almost identical to $-best$ case.

The experimental results are relevant to the rebalancing of the tree-structure state. There is almost no need for rebalancing the tree in $-best$, while in $-worst$ the tree needs to be rebalanced almost each time an element reaches. In Kerschbaum's scheme, the recoding frequency is very high in $-worst$ and very few in $-best$ and $-natural$, since it is in accordance with the rebalancing frequency. Our coding tree unbinds the relationship between the tree rebalancing and the recoding and only triggers the recoding when there is no available encoding for the new ciphertext. Therefore, our recoding frequency is much lower even in $-worst$. It demonstrates that our coding tree makes the recoding condition in our scheme rarer than that in other works, which is mentioned in Section 5.1.

**Table 4: Recoding Frequency on Real Dataset**

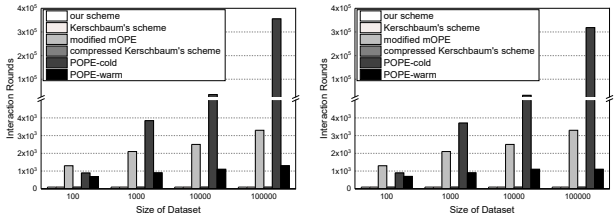
| Case | our scheme | | K's scheme | | modified mOPE | |
|---|---|---|---|---|---|---|
| | Job Title | Other Pay | Job Title | Other Pay | Job Title | Other Pay |
| $-best$ | 0 | 0 | 0 | 0 | 5.0e+6 | 5.1e+6 |
| $-worst$ | 1.1e+6 | 1.8e+6 | 3.8e+7 | 1.8e+8 | 7.3e+6 | 7.7e+6 |
| $-natural$ | 0 | 0 | 0 | 0 | 5.0e+6 | 5.3e+6 |

## 8.5 Interaction

We evaluate the communicational cost of our scheme by performing a number of insertions/queries and counting the interaction rounds.

In our scheme, an insert operation is conducted by our UDF procedure "*PRO_INSERT*". Table 5 depicts the interaction rounds of inserting the values of two columns of the whole real dataset. In any case, there is almost no need for interactively locating ciphertexts on the server side of our scheme and Kerschbaum's scheme, such that the number of interaction rounds is in 1 for each encryption operation. Moreover, in any case, the modified mOPE scheme costs many rounds (i.e., in $O(\log n)$) for ensuring that each ciphertext is stored in order. The experimental results show that our scheme outperforms the modified mOPE scheme due to finishing insertions without additional interaction.

Our scheme conducts the range query by the sql "*select*". To test the interaction rounds in range queries, we randomly pre-store some ciphertexts encrypted from two columns of the real dataset and perform 100 times range queries. There are two start settings: a cold start and a warm start. The default setting is cold-start, while the ciphertexts in the queries have already sorted in the server in a warm-start setting. Only POPE shows different performances in different start settings. Figure 8 depicts the experimental results, where the number of pre-stored ciphertexts varies from 100 to 100000. The modified mOPE scheme costs several interaction rounds for deciding the order of each queried element in the server state. POPE-cold




(a) Interaction Rounds for "Job Title" (b) Interaction Rounds for "Other Pay"

**Figure 8: Interaction Rounds of Range Query**

**Table 5: Interaction Rounds of Insertion**

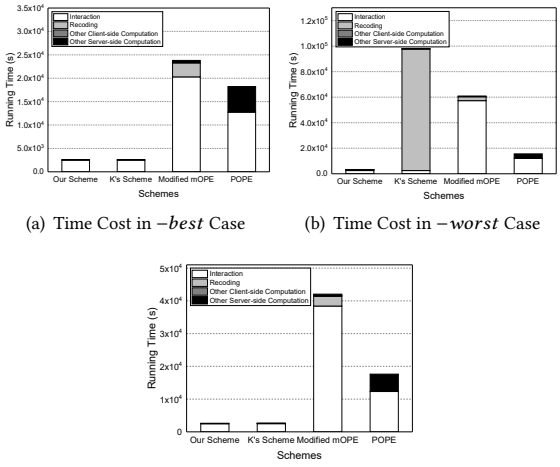
| Case | our scheme | | K's scheme | | modified mOPE | |
|---|---|---|---|---|---|---|
| | Job Title | Other Pay | Job Title | Other Pay | Job Title | Other Pay |
| $-best$ | 247697 | 247697 | 247697 | 247697 | 1636750 | 1641595 |
| $-worst$ | 247697 | 247697 | 247697 | 247697 | 1735330 | 1758638 |
| $-natural$ | 247697 | 247697 | 247697 | 247697 | 1656260 | 1624138 |

involves huge interaction cost since it will suffer a large penalty on the first query. Although POPE-warm may achieve one round interaction for some query, it still spends considerable overheads since it does not constantly store any order-preserving encodings for the range comparison. The performance of our scheme is much better than that of the modified mOPE and POPE. The reason is that our scheme can correctly locate the position of a queried ciphertext, so that we only need 1 rather than $O(\log n)$ interactions.

## 8.6 Overall Application

Finally, we make an overall evaluation of our implementation in the encrypted database. To make fair comparisons between our scheme and others, we set up each scheme for an outsourced encrypted database in the real-world application and observe its performance. As shown in Section 8.1, we rent the high-performance cloud service to eliminate the influence of the unexpected network delay. In each case ($-best$, $-worst$, and $-natural$), we take 100000 records from the column "Job Title" of the real dataset as the input sequence of an experiment. After running a scheme in the experiment, we simulate the real-world scenario by performing insertions and range queries in the database alternately. In more details, 1) the client encrypts 1000 records in turn and submits them to the server-side database, until all ciphertexts have been stored; 2) the client immediately initiates 4 range queries on the database after each submission.

The experimental results are shown in Figure 9. The total running time cost of each scheme is stacked by four parts: *Interaction*, *Recoding*, *Other Client-side Computation*, and *Other Server-side Computation*. Note that *Other Client-side Computation* refers to the time cost on client-side computations other than recoding operations, so as to *Other Server-side Computation*. As a client-state scheme, Kerschbaum's scheme is communication-efficient, but spends extreme time on *Recoding* in $-worst$. As a server-state scheme, the modified mOPE spends most time on *Interaction* and *Recoding*, and has a stable performance in each case. POPE is actually warm-start




(a) Time Cost in $-best$ Case (b) Time Cost in $-worst$ Case

(c) Time Cost in $-natural$ Case

**Figure 9: Time Cost in Real-world Application**

in our experiments, which costs much time on *Interaction* and *Other Client-side Computation*. The computational cost is mainly for adjusting the server state without any order-preserving encoding after a range query. The experimental results are in accordance with the results in Section 8.4 and Section 8.5.

We can learn that in any case, *Interaction* almost dominates the time cost in each scheme, followed by *Recoding*. This trend will become more obvious if the client user is equipped with an elementary bandwidth. Under this situation, our scheme outperforms other schemes due to its outstanding performance in *Interaction* and *Recoding*. That means our design goals "without additional interactions" and "low recoding frequency" in Section 4 are very important for improving the performance of a FH-OPE scheme. The experimental results demonstrate that our scheme is practical for real-world database applications.

# 9 CONCLUSION

In this paper, we propose the first FH-OPE scheme achieves the small client storage of size $O(N)$ and 1 interaction per insert/query operation. We propose a new coding tree with the coding strategy that reduces the frequency of encoding updates.

Experimental results show that our scheme costs the smaller client storage when storing the same number of plaintexts as existing client-state schemes. Our scheme costs less communications when inserting/querying with the same number of queries as existing server-state schemes. Moreover, the frequency of encoding updates in our scheme is less than that in existing works when inserting the same number of plaintexts in each case. Our evaluation demonstrates that the remarkable performance above makes our scheme practical for real-world database applications.


# ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China (No. 2020YFB1005700), National Natural Science Foundation of China (No. 62032012 and No. 61802078).

## REFERENCES


[1] Saleh Ahmed, Annisa, Asif Zaman, Zhan Zhang, Kazi Md. Rokibul Alam, and Yasuhiko Morimoto. 2019. Semi-Order Preserving Encryption Technique for Numeric Database. *Int. J. Netw. Comput.* 9, 1 (2019), 111–129.

[2] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. 2018. The Tao of Inference in Privacy-Protected Databases. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1715–1728. https://doi.org/10.14778/3236187.3236217

[3] Dmytro Bogatov, George Kollios, and Leonid Reyzin. 2019. A Comparative Evaluation of Order-Revealing Encryption Schemes and Secure Range-Query Protocols. *Proceedings of the VLDB Endowment* 12, 8 (2019), 933–947. https://doi.org/10.14778/3324301.3324309

[4] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings (Lecture Notes in Computer Science)*, Antoine Joux (Ed.), Vol. 5479. Springer, Springer, 224–241. https://doi.org/10.1007/978-3-642-01001-9_13

[5] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, Phillip Rogaway (Ed.), Vol. 6841. Springer, Springer, 578–595. https://doi.org/10.1007/978-3-642-22792-9_33

[6] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. 2015. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9057. Springer, 563–594. https://doi.org/10.1007/978-3-662-46803-6_19

[7] David Cash, Feng-Hao Liu, Adam O'Neill, and Cong Zhang. 2016. Reducing the Leakage in Practical Order-Revealing Encryption. *IACR Cryptol. ePrint Arch.* 2016 (2016), 661. http://eprint.iacr.org/2016/661

[8] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. 2016. Practical Order-Revealing Encryption with Limited Leakage. In *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, Thomas Peyrin (Ed.), Vol. 9783. Springer, Springer, 474–493. https://doi.org/10.1007/978-3-662-52993-5_24

[9] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2011. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security* 19, 5 (2011), 895–934.

[10] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos N. Garofalakis. 2016. Practical private range search revisited. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 185–198. https://doi.org/10.1145/2882903.2882911

[11] Michael Egorov and MacLane Wilkison. 2016. ZeroDB white paper. abs/1602.07168. arXiv:1602.07168 http://arxiv.org/abs/1602.07168

[12] Jieun Eom, Dong Hoon Lee, and Kwangsu Lee. 2018. Multi-Client Order-Revealing Encryption. *IEEE Access* 6 (2018), 45458–45472. https://doi.org/10.1109/ACCESS.2018.2864991

[13] Benny Fuhry, Florian Kerschbaum, et al. 2020. Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves. *CoRR* abs/2002.05097 (2020). https://arxiv.org/abs/2002.05097

[14] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 315–331. https://doi.org/10.1145/3243734.3243864

[15] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-Abuse Attacks against Order-Revealing Encryption. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE, IEEE Computer Society, 655–672. https://doi.org/10.1109/SP.2017.44

[16] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 361–378. https://doi.org/10.1145/3319535.3363210

[17] Helene Haagh, Yue Ji, Chenxing Li, Claudio Orlandi, and Yifan Song. 2017. Revealing Encryption for Partial Ordering. In *Cryptography and Coding - 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings*, Máire O'Neill (Ed.), Vol. 10655. Springer, 3–22. https://doi.org/10.1007/978-3-319-71045-7_1

[18] Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2010. MV-OPES: Multivalued-order preserving encryption scheme: A novel scheme for encrypting integer value to many different values. *IEICE TRANSACTIONS on Information and Systems* 93, 9 (2010), 2520–2533. https://doi.org/10.1587/transinf.E93.D.2520

[19] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. 2016. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 1329–1340. https://doi.org/10.1145/2976749.2978386

[20] Florian Kerschbaum. 2015. Frequency-hiding order-preserving encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. ACM, 656–667. https://doi.org/10.1145/2810103.2813629

[21] Florian Kerschbaum and Anselme Tueno. 2019. An efficiently searchable encrypted data structure for range queries. In *European Symposium on Research in Computer Security (Lecture Notes in Computer Science)*, Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan (Eds.), Vol. 11736. Springer, Springer, 344–364. https://doi.org/10.1007/978-3-030-29962-0_17

[22] Kee S. Kim. 2019. New Construction of Order-Preserving Encryption Based on Order-Revealing Encryption. *J. Inf. Process. Syst.* 15, 5 (2019), 1211–1217. http://www.jips-k.org:80/q.jips?cp=pp&pn=715

[23] Marie-Sarah Lacharité and Kenneth G. Paterson. 2018. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology* 2018, 1 (2018), 277–313. https://doi.org/10.13154/tosc.v2018.i1.277-313

[24] Kevin Lewi and David J. Wu. 2016. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1167–1178. https://doi.org/10.1145/2976749.2978376

[25] Dongxi Liu and Shenlu Wang. 2012. Programmable Order-Preserving Secure Index for Encrypted Database Query. In *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, Rong Chang (Ed.). IEEE, IEEE Computer Society, 578–595. https://doi.org/10.1109/CLOUD.2012.65

[26] Zhe Liu, Kim-Kwang Raymond Choo, and Minghao Zhao. 2017. Practical-oriented protocols for privacy-preserving outsourced big data analysis: Challenges and future research directions. *Computers & Security* 69 (2017), 97–113. https://doi.org/10.1016/j.cose.2016.12.006

[27] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. 2015. Modular Order-Preserving Encryption, Revisited. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 763–777. https://doi.org/10.1145/2723372.2749455

[28] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 644–655. https://doi.org/10.1145/2810103.2813651

[29] Rishabh Poddar, Tobias Boelter, and Raluca A. Popa. 2019. Arx: an encrypted database using semantically secure encryption. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1664–1678. https://doi.org/10.14778/3342263.3342641

[30] Raluca Ada Popa, Frank H Li, and Nickolai Zeldovich. 2013. An ideal-security protocol for order-preserving encoding. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley,CA, USA, May 19-22, 2013*. IEEE, IEEE Computer Society, 463–477. https://doi.org/10.1109/SP.2013.38

[31] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2012. CryptDB: processing queries on an encrypted database. *Commun. ACM* 55, 9 (2012), 103–111. https://doi.org/10.1145/2330667.2330691

[32] David Pouliot, Scott Griffy, and Charles V. Wright. 2017. The Strength of Weak Randomization: Efficiently Searchable Encryption with Minimal Leakage. *IACR Cryptol. ePrint Arch.* 2017 (2017), 1098. http://eprint.iacr.org/2017/1098

[33] Daniel S Roche, Daniel Apon, Seung Geol Choi, and Arkady Yerukhimovich. 2016. POPE: Partial order preserving encoding. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1131–1142. https://doi.org/10.1145/2976749.2978345