Secure and Practical Functional Dependency Discovery in Outsourced Databases

Xinle Cao, Zhejiang University, Hangzhou, China, xinle@zju.edu.cn

Yuhan Li, Zhejiang University, Hangzhou, China, yuhan2165@zju.edu.cn

Dmytro Bogatov[§], Amazon Inc., Boston, MA 02210, bogatov@amazon.com

Jian Liu[§], Zhejiang University, Hangzhou, China, liujian2411@zju.edu.cn

Kui Ren, Zhejiang University, Hangzhou, China, kuiren@zju.edu.cn

***Abstract*—The popularity of cloud computing has made outsourced databases prevalent in real-world applications. To protect data security, numerous encrypted outsourced databases have been proposed for this paradigm. However, the maintenance of encrypted databases has scarcely been addressed. In this paper, we focus on a typical maintenance task — functional dependency (FD) discovery. We develop novel FD protocols in encrypted databases while guaranteeing minimal leakages: nothing is revealed besides the database size and the actual discovered FDs. As far as we know, we are the first to formally define *secure FD discovery with minimal leakage*.**

We present two oblivious FD protocols and prove them secure in the presence of the persistent adversary (monitoring processes on the server). The first protocol leverages Oblivious RAM (ORAM) and is suitable for dynamic databases. The second protocol relies on oblivious sorting and is more practical in static databases due to high parallelism. We also present a thorough experimental evaluation of the proposed methods.

***Index Terms*—encrypted databases, data management, functional dependency discovery, obliviousness**

I. Introduction

With the popularity of cloud computing and the growth of data volume, an outsourced database has become an important and practical paradigm: the client $\mathcal{C}$ uploads its large database to a cloud server $\mathcal{S}$ for storage and management. $\mathcal{C}$ accesses and utilizes the database by directly issuing queries to $\mathcal{S}$. In this way, $\mathcal{C}$ does not need to purchase expensive equipment and manage it themselves. $\mathcal{C}$ can also elastically adjust the cloud resource rented from $\mathcal{S}$ according to its business demand.

Although an outsourced database provides multiple advantages, it also poses a risk to data security as now possibly untrusted $\mathcal{S}$ can observe the database. To this end, plenty of encrypted databases [5], [13], [29], [36], [46], [54], [50] have been proposed, which encrypt the database before uploading. They adopt some advanced cryptographic techniques [44], [55] and secure enclaves [13] to allow database operations while avoiding the leakage of sensitive information. Encrypted outsourced database has become a promising direction in the database community, and many works have been presented to solve query processing in encrypted databases such as range queries [5], [29], [45], [44] and join queries [10], [27], [53], [47].

A. Database Maintenance

Despite rich work on query processing, there is little work on database maintenance in encrypted databases [14], [30]. Database maintenance generally cannot be performed via query interface as that can result in unexpected leakages [30]. This issue makes most of the existing encrypted databases [13], [19], [29], [44] lack solutions for database maintenance with formal security guarantees. Database maintenance, however, is very important in data management. It guarantees the database is organized reasonably, the data quality is promoted, and the query processing is well-optimized.

A typical task in database maintenance is *functional dependency* (FD) discovery [32]. Given two attribute sets A, B, an FD: $A \rightarrow B$ constraint means that the values of A uniquely determine the values of B. For example, the FD: *Zipcode* $\rightarrow$ *City* implies that all records with the same *Zipcode* values have the same *City* values. FD reveals the relationships between different attribute sets and is a typical type of data dependency. With FDs in the database, we can solve multiple problems in database maintenance including schema normalization [12], data cleaning [16], and database (re)design [57]. Here we give an application scenario of an FD in query optimization to show why FD discovery is important in encrypted databases.

Example. *Consider an `employee` table that preserves records for employees in the company. Each record stores the `Position` and `Department` of an employee. If there is an FD : `Position` $\rightarrow$ `Department`, then we can identify only the `Position` value to retrieve required records instead of both `Position` and `Department`. We note that in encrypted databases in particular, reducing two equality tests for attributes to one is significant, e.g., half costs can be reduced in [44].*

[§]This work is not related to the work of Dmytro Bogatov in Amazon.

[§]Jian Liu is the corresponding author.

This work focuses on FD discovery in encrypted databases as the necessary prerequisite for database maintenance.

B. Secure FD Discovery

The FD discovery in encrypted databases is challenging because of an extra requirement on minimal leakage. As a crucial task in data management, FD discovery has been studied by [16], [23], [32], [52] in the last decades. Nonetheless, it is far from trivial to apply these approaches in encrypted databases because these approaches do not consider data security. Dong et al. [14] are the first to present FD discovery in encrypted databases with a formal security guarantee. But they enable FD discovery at the cost of the leakage of well-defined partial frequency information about plaintexts. As the frequency leakage has been proven extremely dangerous to $\mathcal{C}$ [20], [39], this leakage may be unacceptable.

In this paper, we require protocols for FD discovery to leak minimal information to an adversary. Following existing literature [10], [13], [36], we assume a *persistent adversary* [5], [44] who is honest-but-curious and can observe everything available in $\mathcal{S}$ during the whole execution of protocols for FD discovery. Now we intuitively define secure FD discovery in encrypted databases:

Secure FD discovery protocol leaks minimal information necessary to obtain the result (leaks only the database size and discovered FDs) under the persistent adversary.

The leakage of size is generally accepted in encrypted databases [44], [10], [27] and the leakage of discovered FDs is necessary to complete the task, therefore we refer to them as *the minimal leakages* of secure FD discovery.

To achieve secure FD discovery in the presence of persistent adversary, we have to avoid leakage from not only ciphertexts (i.e., snapshot) but also access patterns — the patterns of memory accesses [24], [26], [28]. We follow prior works [50], [29] to apply semantically secure encryption to conceal the leakage from ciphertexts. The more challenging part is to mitigate the leakage from access patterns, which requires the whole process of FD discovery to be oblivious. That is, the access patterns should only depend on the database size. To this end, we apply cryptographic primitives *Oblivious RAM* (ORAM) and *oblivious sorting* for performing FD discovery obliviously. We note that the "oblivious" primitives can still be used insecurely. For example, the number of accesses to ORAM protocols can leak sensitive information [10], [5]. Extra care is needed to integrate the primitives into a system and prove it is oblivious [9], [10].

C. Contributions

To summarize, this paper makes the following contributions:

- The first security definition for FD discovery in secure outsourced databases requiring minimal leakage.
- A flexible ORAM-based secure FD discovery protocol applicable to both static and dynamic databases.
- A practical secure FD discovery protocol with oblivious sorting for static databases that exhibits high parallelism.
- The implementation and experimental evaluation of the two methods in the cloud setting.

II. Preliminaries

Notations. For a set I, $|I|$ denotes the cardinality of I, i.e., the number of elements in I. For a positive integer k, we use $[k]$ to denote the contiguous integer set $\{0, 1, 2, ..., k-1\}$. Let $\mathbf{1}\{b\}$ returns an integer 1 if b is *true* and 0 otherwise. We denote the security parameter as λ.

A. Outsourced Database

Without loss of generality, we suppose a client $\mathcal{C}$ owns a database DB with n rows and m attributes (columns) where each row represents an individual record. We denote the value of record r in attribute X as $r[X]$. In particular, we assume each record r in DB can be represented or mapped by a unique number denoted by $r[ID]$, like their row numbers. In data outsourcing, $\mathcal{C}$ encrypts DB to $\widehat{\mathsf{DB}}$ and uploads $\widehat{\mathsf{DB}}$ to a cloud server $\mathcal{S}$. To access $\widehat{DB}$, $\mathcal{C}$ issues queries and lets $\mathcal{S}$ execute them. In particular, it is possible that $\mathcal{C}$ will modify the database by issuing insertion and deletion queries, i.e., $\widehat{\mathsf{DB}}$ can be dynamic. Similar to prior works [15], [44], [58], we assume the encryption is conducted at *a cell level*: each attribute value in a record is encrypted individually.

Note: This work assumes a fairly generic outsourced database (individually accessible symmetrically encrypted record values). Also, $\mathcal{C}$ and $\mathcal{S}$ do not need query processing session to run FD protocols nor do the FD protocols conflict with query processing. This way the proposed FD discovery can run in *online* and *offline* settings, similar to the works in secure multiparty computation (MPC) [37] and private information retrieval (PIR) [35]. Therefore, the proposed methods are highly applicable to existing outsourced database designs.

B. Functional Dependency Discovery

Given two attribute sets A and B in database DB, there exists an FD F between A and B, denoted by $F : A \rightarrow B$ iff for any pair of records r_1, r_2 in DB, if $r_1[A] = r_2[A]$, then $r_1[B] = r_2[B]$. For a functional dependency F, we denote the attribute sets at its left and right hand as $LHS(F)$ and $RHS(F)$, respectively. Note $LHS(F)$ and $RHS(F)$ represent attribute sets, so they may consist of single or multiple attributes, e.g., $F : \{\textit{Course_ID}, \textit{Semester}\} \rightarrow \{\textit{Professor}, \textit{Classroom}\}$.

There has been a line of work [32], [41], [42] showing how to discover all FDs in DB efficiently without encryption and security. They effectively solve two problems: (1) given two attribute sets A and B in DB, how to validate if $F : A \rightarrow B$ holds; (2) given a database DB, how to plan the validation on different attribute sets to discover all FDs efficiently, e.g., if $F_1 : A \rightarrow B$ and $F_2 : A \rightarrow C$ have been shown to hold, then $F_3 : A \rightarrow B \cup C$ must hold and does not need to be validated. In this paper, we focus on the first problem because the validation plan on DB is natural following the prior works [32], [23].

C. Partition-based Methods

Given an attribute set X, two records r, r' in database DB are *equivalent* if $r[X] = r'[X]$. We denote the *equivalent class* of a record r with attribute set X as $[r]_X := \{r' \in \mathsf{DB} | r'[X] = r[X]\}$. Now records in DB can be classified according to their equivalence class. The set $\pi_X = \{[r]_X | r \in \mathsf{DB}\}$ is called a *partition* of DB under X. The partition consists of multiple disjoint sets and each set has a unique value on attribute set X for the records in it. The partitions under different attribute sets can be used to validate FDs with the following theorem [32].

Theorem 1. *Given any two attribute sets A, B, an FD $F : A \rightarrow B$ holds iff $|\pi_A| = |\pi_{A \cup B}|$.*

This theorem is widely known and used in prior works on FD discovery [32], [40], [61]. These partition-based methods are typical and efficient solutions for FD discovery. Here we give an example in Fig. 1 to show how this theorem works. In the example, we can calculate that $|\pi_{\{\mathit{Name}\}}| = |\pi_{\{\mathit{Name,\ City}\}}|$ and $|\pi_{\{\mathit{Name}\}}| \neq |\pi_{\{\mathit{Name,\ Birth}\}}|$. So we know *Name* $\rightarrow$ *City* holds, but *Name* $\rightarrow$ *Birth* does not. In this paper, we also follow this theorem and calculate the partitions securely to achieve secure FD discovery.

	Name	*City*	*Birth*
r_1	Alice	Boston	Jan
r_2	Bob	Boston	May
r_3	Bob	Boston	Jan
r_4	Carol	New York	Sep

Fig. 1. An example for Theorem 1. It can be calculated that $\pi_{\{\mathit{Name}\}} = \pi_{\{\mathit{Name,City}\}} = \{\{r_1\}, \{r_2, r_3\}, \{r_4\}\}$ and $\pi_{\{\mathit{Name,Birth}\}} = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}\}$.

III. Problem Overview

A. Problem Definition

The *encrypted database system* consists of two entities: the client $\mathcal{C}$ and the server $\mathcal{S}$. The client has a database with n rows and m attributes whose set is denoted by $T := \{T_1, T_2, ..., T_m\}$. For data outsourcing, $\mathcal{C}$ encrypts DB to $\widehat{\mathsf{DB}}$ and uploads $\widehat{\mathsf{DB}}$ to $\mathcal{S}$. To achieve FD discovery, $\mathcal{C}$ and $\mathcal{S}$ interact with each other to run a protocol Π on $\widehat{\mathsf{DB}}$ and output a set S consisting of pairs such that

$$(A, B) \in S \iff A, B \subset T \text{ and } A \rightarrow B.$$

Note $\mathcal{C}$ has only limited resources (including small memory), e.g., personal computers, wearables, so it cannot download the whole database to discover FDs locally. $\mathcal{C}$ has to interact with $\mathcal{S}$ to complete the FD discovery revealing the access patterns on $\widehat{\mathsf{DB}}$ to $\mathcal{S}$ in the process, leaking sensitive information about DB. This setting is common in recent encrypted databases [10], [27] and is especially consistent with those based on hardware enclaves [13], [21], [36] as the enclave can also be regarded as such client. Therefore, the protocols proposed in this paper can be easily integrated with most existing encrypted databases to achieve database maintenance.

B. Security Model

In this paper, we consider the server $\mathcal{S}$ an honest-but-curious *persistent adversary*. It follows the predefined protocols Π as expected to discover FDs. But it tries to get as much sensitive information as possible from its view during the execution. Before we introduce the security notion, we define the *minimal leakage function* of FD discovery on DB

$$\mathcal{L}(\mathsf{DB}) := \{\mathsf{Size}(\mathsf{DB}), \mathsf{FD}(\mathsf{DB})\}$$

where $\mathsf{Size}(\mathsf{DB})$ is the database size (m, n) and $\mathsf{FD}(\mathsf{DB})$ is the set of all FDs in DB. We call it the minimal leakage function because there is no other leakage besides the database size $\mathsf{Size}(\mathsf{DB})$ and functional dependency $\mathsf{FD}(\mathsf{DB})$. The size information is commonly leaked in encrypted databases [13], [44], [54] and the FD leakage is necessary to do database maintenance. Now we can formally define the secure FD discovery protocol Π. Following prior works [11], [44], we adopt a simulation-based security notion. We assume a passive adversary $\mathcal{A}$ that has all the view of $\mathcal{S}$ and wants to distinguish the real world ($\mathbf{Real}$) and the ideal world ($\mathbf{Ideal}$). In the real world, the protocol Π is conducted on $\widehat{\mathsf{DB}}$ while in the ideal world, given the access of the leakages $\mathcal{L}(\mathsf{DB})$, a simulator Sim simulates the execution of Π on $\widehat{\mathsf{DB}}$.

Definition 1 (Secure FD discovery)**.** *For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ that has all the views of $\mathcal{S}$, we say that Π is a secure FD discovery protocol if there exists a PPT simulator* Sim *such that*

$$|\Pr[\mathbf{Real}^{\Pi}_{\mathcal{A}} = 1] - \Pr[\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathsf{Sim},\mathcal{L}} = 1]| \leq \mathsf{negl}(\lambda)$$

where λ is the security parameter and $\mathsf{negl}(\lambda)$ *is a negligible function in λ.*

The security analysis and proof of our protocols are available in Section VI.

C. Obliviousness

To achieve secure FD discovery, we have to conceal the leakage from ciphertexts and access patterns. Similar to prior works [13], [27], we adopt semantically secure encryption which avoids any leakage about plaintexts (besides the plaintext length) from ciphertexts. The decryption of ciphertexts only happens inside $\mathcal{C}$. We also let $\mathcal{C}$ apply re-encryption to guarantee the ciphertexts read and written are distinct.

The challenge is to mitigate the leakage from access patterns [24], [26], [28] in $\mathcal{S}$, which requires *oblivious algorithms*. That is, the distribution of access patterns in $\mathcal{S}$ is dependent on only the database size and has no relation to database contents. Formally, we define oblivious algorithms in encrypted databases as below:

Definition 2 (Oblivious algorithm)**.** *For any two databases* DB_0 *and* DB_1 *with the same size, denote the access patterns in* $\mathcal{S}$ *of running algorithm* $\mathcal{P}$ *on* DB_0 *and* DB_1 *as* $\mathsf{Trace}(\mathsf{DB}_0, \mathcal{P})$ *and* $\mathsf{Trace}(\mathsf{DB}_1, \mathcal{P})$*, respectively.* $\mathcal{P}$ *is an oblivious algorithm if* $\mathsf{Trace}(\mathsf{DB}_0, \mathcal{P})$ *and* $\mathsf{Trace}(\mathsf{DB}_1, \mathcal{P})$ *are computationally indistinguishable.*

In this paper, we introduce two oblivious algorithms for calculating partitions of any attribute set X. They are used to construct secure FD protocols. One is based on *oblivious sorting* while the other applies the well-known *Oblivious RAM* (ORAM), used in the recent works [10], [27] for designing oblivious algorithms in encrypted databases.

a) Oblivious sorting: Oblivious sorting is widely used in encrypted databases and has attracted a lot of research interest [1], [7], [48]. There exist some oblivious sorting algorithms [18] with only $O(n \log n)$ complexity and Lin et al. [31] have proven that this complexity is the lower bound. Unfortunately, these optimal algorithms are either inefficient because of the large constant overhead or cannot be run in parallel, making them impractical. In this paper, we follow prior works [13], [27] and adopt *bitonic sorting* [4]. Although this sorting requires $O(n \log^2 n)$ computational complexity, it is efficient in practice and can achieve high parallelism. As this sorting will be used like a black box in our algorithms, we refer to [4] for more details. We parameterize the oblivious sorting with the following definition.

Definition 3 (Oblivious sorting)**.** *Consider a database* DB *consisting of* n *records and* m *attributes, given a positive integer* $k \leq m$*, we define*

$$\mathsf{DB}' \leftarrow \mathsf{ObliviousSort}(attr_k, \mathsf{DB})$$

which is an oblivious algorithm that inputs the database DB*, and returns a new database* DB' *that consists of the same* n *records as* DB *but has them in an order such that*

$$\forall i, j \in [n], i < j \iff \mathsf{DB}'[i][attr_k] \leq \mathsf{DB}'[j][attr_k].$$

Here we assume that the data in each cell orderable.

b) Oblivious RAM: ORAM is a well-known cryptographic protocol used to allow $\mathcal{C}$ to access a record from $\widehat{\mathsf{DB}}$ obliviously. However, algorithms using ORAM to access data are not necessarily oblivious. For example, the number of accesses ORAM makes can reveal some sensitive information [5], [10]. Therefore, algorithms with ORAM still need to be designed carefully to make them oblivious and efficient. In this paper, we adopt one of the most simple and efficient ORAM constructions named PathORAM [55]. We apply the non-recursive PathORAM (like [5], [10]) to improve the efficiency. We also use ORAM in a black-box fashion and define the interfaces for calling ORAM as below.

Definition 4 (ORAM)**.** *An ORAM protocol consists of three oblivious subprotocols* $(\mathsf{Setup}, \mathsf{Read}, \mathsf{Write})$*.* $\mathcal{C}$ *and* $\mathcal{S}$ *interact with each other to run the following subprotocols.*

- $(\mathsf{st}, \mathcal{O}) \leftarrow \mathsf{Setup}(1^\lambda)$*: This protocol takes the security parameter* λ *as input, and outputs a secret state* st *for* $\mathcal{C}$ *and an encrypted memory* $\mathcal{O}$ *for* $\mathcal{S}$*.*
- $(value, \mathcal{O}) \leftarrow \mathsf{Read}(key, \mathcal{O})$*: In this protocol,* $\mathcal{C}$ *inputs a key and retrieves the value corresponding to the key. The pair* $(key, value)$ *is stored encrypted in* $\mathcal{O}$ *(if there exists no pair associated with* key*, then* $\perp$ *is returned).* $\mathcal{S}$ *inputs the encrypted memory* $\mathcal{O}$*, updates and outputs it.*
- $\mathcal{O} \leftarrow \mathsf{Write}((key, value), \mathcal{O})$*: In this protocol,* $\mathcal{C}$ *inputs a key-value pair* $(key, value)$ *and gets nothing,* $\mathcal{S}$ *inputs the encrypted memory and gets it updated such that the pair* $(key, value)$ *is stored encrypted in it.*

The protocols Read *and* Write *are mutually indistinguishable for* $\mathcal{S}$*. We write them separately for the ease of presentation.*

IV. Secure FD discovery on static databases

In this section, we consider secure FD discovery on the basic scenario where DB is static. After $\mathcal{C}$ encrypts DB and uploads $\widehat{\mathsf{DB}}$, it will not issue insertion, deletion, or update queries to modify $\widehat{\mathsf{DB}}$. We first clarify the framework for the FD discovery and then introduce an important technique in our algorithms. Finally, we describe our specific algorithms.

A. Framework

The tasks for FD discovery on DB based on partitions can be divided into three levels as below:

- *Attribute-level:* Given an attribute set X in DB, calculate the partition π_X and $|\pi_X|$ (i.e., the number of distinct values under attribute set X).
- *Set-level:* Given two attribute sets (X, Y), check if $|\pi_X| = |\pi_{X \cup Y}|$. The FD: $X \rightarrow Y$ holds iff the equation holds.
- *Database-level:* Given the database DB, determine all attribute sets that need to be checked in the set level.

We illustrate the framework in Fig. 2. In this paper, we will focus on only the attribute-level task because the set-level and database-level tasks are natural and do not leak any

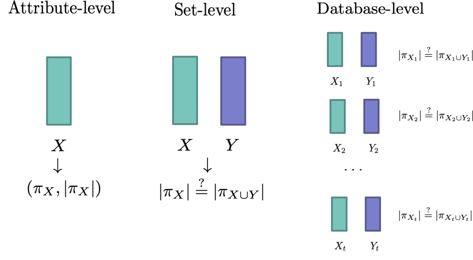


Fig. 2. The framework of partition-based methods.

information besides the minimal leakages we allow. The set-level task can be done by just checking if two variables are equal. The checking results reveal only if the FDs hold. For the database-level task, in the worst case, it can be done by checking every two attribute sets, which is very expensive. We adopt the top-down method (including its pruning rules) in [23], [40], which makes use of discovered FDs to remove some attribute sets from checking. For example, if $F : X \to Y$ does not hold, then $F : X \to Y \cup Z$ also does not hold and should not be checked. This method also leaks nothing else besides discovered FDs. There is an important property in this method, we introduce it here and refer to [23] for more details.

Property 1 (Partition-friendly). *For any attribute set X where $|X| \geq 2$, it is guaranteed that, before calculating π_X, there exists two distinct attribute sets $X_1, X_2 \subsetneq X$ such that $X_1 \cup X_2 = X$ and (π_{X_1},π_{X_2}) have been calculated.*

This property is important for partition-based methods [23], [40] of FD discovery and our secure FD discovery protocols because π_X can be calculated efficiently with (π_{X_1}, π_{X_2}) [23]. In the remaining sections, we will propose two distinct oblivious algorithms to calculate $(\pi_X, |\pi_X|)$ for any attribute set X. Secure FD discovery protocols are constructed by directly combining them with the simple set-level checking and the database-level top-down method.

B. Attribute Compression

We apply the *attribute compression* to make the partition calculation can be done in a constant way. Attribute compression is an important optimization in FD discovery without security. It compresses values under attribute set X when calculating π_X. The compression can effectively accelerate the calculation: it reduces the length of attribute values such that the processing time and I/O cost are smaller [42], especially for the case where $|X|$ is large. Here we give a simple example for illustrating the compression, consider three records under a single attribute $\{Name\}$ like $(r_1[Name], r_2[Name], r_3[Name]) = (Alice, Bob, Bob)$. Map each distinct attribute value to a unique number, the three values can be compressed to $(1, 2, 2)$ while preserving $\pi_{Name} = \{\{r_1\}, \{r_2, r_3\}\}$. Our algorithms also compress attribute values and preserve π_X with the compression.

The prior methods [42], [52] compress each distinct value under X to a unique integer in $[n]$ even if $|X| \geq 2$. However, they cannot be deployed in our methods as they leak sensitive information like plaintext frequency. We adopt a new injective mapping to compress attribute values. For a record r and an attribute set X, we assign a pair denoted by $(key_X, label_X)$ to compress it. We define key_X as

$$key_X = \begin{cases} r[X] \;, & |X| = 1, \\ r[X]^C, & |X| > 1. \end{cases}$$

where $r[X]^C$ is a unique number in $[n^2+n]$ mapped by the value of $r[X]$ and $label_X$ is a unique number in $[n]$ mapped by key_X.

Now we explain the calculation of $r[X]^C$. Recall Property 1 guarantees that before we calculate π_X, there exist two different subsets of X denoted by X_1 and X_2 ($X_1 \cup X_2 = X$) whose partitions have been calculated. As their compression is also conducted when calculating partitions, we now have two pairs for each record r: $(key_{X_1}, label_{X_1})$ and $(key_{X_2}, label_{X_2})$. Then we define

$$r[X]^C := label_{X_1} \cdot n + label_{X_2}.$$

and then map this key_X to a unique $label_X$. In this way, we ensure each distinct value under X is compressed to a unique key_X and also a unique $label_X$. The length of key_X is the same as $r[X]$ when $|X| = 1$ and no more than $2\lceil \log n \rceil + 1$ when $|X| \geq 2$. The length of $label_X$ is always no more than $\lceil \log n \rceil + 1$ no matter what $|X|$ is. We illustrate the compression with an example in Fig. 3. An important fact is that the partition calculation for two attribute sets X and Y can be identical even if $|X| = 2$ and $|Y| = 100$ because $(key_X, label_X)$ and $(key_Y, label_Y)$ have the same length. This fact allows our algorithms to calculate π_X when $|X| \geq 2$ constantly. The cost does not increase with the increase of $|X|$.

	X_1	X_2
r_1	Alice	Jan
r_2	Bob	Jan
r_3	Bob	Jan
r_4	Carol	Sep

(a) Database DB

	$label_{X_1}$	$label_{X_2}$	key_X
r_1	1	1	5
r_2	2	1	9
r_3	2	1	9
r_4	3	2	14

(b) Compressed values ($X = X_1 \cup X_2$)

Fig. 3. An example showing our attribute compression ($key_X = label_{X_1} \cdot 4 + label_{X_2}$). To calculate π_X, take r_1 as an example, we input 5 instead of the long union string of *Alice* and *Jan*.

C. ORAM-based Oblivious Algorithm

In this section, we propose the oblivious algorithm based on ORAM for calculating partitions. Besides static

Algorithm 1: Calculate $|\pi_X|$ for a single attribute

Input: $\{(r_i[X], r_i[id])\}_{i=1}^n$
Output: Integer $|\pi_X|$, ORAMs $(\mathcal{O}_X^{\mathsf{KL}}, \mathcal{O}_X^{\mathsf{IL}})$

```
   /* Initialize ORAMs */
1  O_X^KL ← Setup(1^λ), O_X^IL ← Setup(1^λ)
2  card_X = 0

   /* Establish ORAMs */
3  for i = 1; i ≤ n; i++ do
4  |   key_X = r_i[X]
5  |   (label_X, O_X^KL) ← Read(key_X, O_X^KL)
6  |   flag = 1{label_X ≠ ⊥}
7  |   label_X = flag · label_X + (1 − flag) · card_X
   |
8  |   O_X^IL ← Write((r_i[ID], label_X), O_X^IL)
9  |   O_X^KL ← Write((key_X, label_X), O_X^KL)
10 |   card_X = card_X + (1 − flag)
11 end

12 Return |π_X| := card_X, π_X := (O_X^KL, O_X^IL)
```

databases, it allows $\mathcal{C}$ to insert new records. It is flexible as it can be extended to dynamic databases with both insertion and deletion, which we will show in Section V.

a) Setup: For any attribute set X, the value of $|\pi_X|$ is equivalent to the number of distinct values of $r[X]$ in DB. Recall in Section IV-B, we introduced our approach for compressing attribute value $r[X]$ to a unique key_X and a unique $label_X$ for any attribute set X. Thus we can calculate $|\pi_X|$ by counting the distinct values of key_X. We establish two ORAMs to achieve the calculation:

- **Key-Label ORAM $\mathcal{O}_X^{\mathsf{KL}}$.** This ORAM is designed to count the number of distinct key_X. For each distinct key_X, this ORAM stores the pair $(key_X, label_X)$. It records $label_X$ to guarantee each key_X corresponds to a unique $label_X$. When this ORAM has been established, the number of pairs in it is the value of $|\pi_X|$.
- **ID-Label ORAM $\mathcal{O}_X^{\mathsf{IL}}$.** For each record r, this ORAM stores a pair $(r[ID], label_X)$ where $r[ID]$ is the unique representation of record r and $label_X$ is the unique value corresponding to $r[X]$. This ORAM is not used in calculating π_X but is necessary for the calculation of π_Y for any attribute set Y such that $X \subsetneq Y$. Also, we note this ORAM preserves π_X as it stores $r[ID]$.

We show the detailed algorithms for establishing the two ORAMs in Algorithm 1 and 2. The algorithms are designed in an oblivious style [27]: instead of *If-else*, all variables are accessed independently of their values (e.g., line 7 in Algorithm 1). Notably, $\mathcal{C}$ and $\mathcal{S}$ interact with each other to conduct the algorithms. It is required that only $\mathcal{C}$ can decrypt

Algorithm 2: Calculate $|\pi_X|$ for multiple attributes

Input: $(\mathcal{O}_{X_1}^{\mathsf{KL}}, \mathcal{O}_{X_1}^{\mathsf{IL}})$, $(\mathcal{O}_{X_2}^{\mathsf{KL}}, \mathcal{O}_{X_2}^{\mathsf{IL}})$
Output: Integer $|\pi_X|$, ORAMs $(\mathcal{O}_X^{\mathsf{KL}}, \mathcal{O}_X^{\mathsf{IL}})$

```
   /* Initialize ORAMs */
1  O_X^KL ← Setup(1^λ), O_X^IL ← Setup(1^λ)
2  card_X = 0

   /* Establish ORAMs */
3  for i = 1; i ≤ n; i++ do
   |   /* Construct key_X */
4  |   (label_X1, O_X1^IL) ← Read(r_i[ID], O_X1^IL)
5  |   (label_X2, O_X2^IL) ← Read(r_i[ID], O_X2^IL)
6  |   key_X = label_X1 · n + label_X2
   |
7  |   (label_X, O_X^KL) ← Read(key_X, O_X^KL)
8  |   flag = 1{label_X ≠ ⊥}
9  |   label_X = flag · label_X + (1 − flag) · card_X
   |
10 |   O_X^IL ← Write((r_i[ID], label_X), O_X^IL)
11 |   O_X^KL ← Write((key_X, label_X), O_X^KL)
12 |   card_X = card_X + (1 − flag)
13 end

14 Return |π_X| := card_X, π_X := (O_X^KL, O_X^IL)
```

ciphertexts and see variable values. Although $\mathcal{S}$ knows the whole algorithms, it can only observe variable ciphertexts and transfer them to $\mathcal{C}$. For example, in line 4 of Algorithm 1, $\mathcal{S}$ transfers the ciphertext of $r_i[X]$ to $\mathcal{C}$ while $\mathcal{C}$ decrypts it to assign key_X with the value. This design paradigm is widely adopted in encrypted databases [10], [45], [50].

b) Calculation: The ORAM establishment for a single attribute X (i.e., $|X| = 1$) is shown in Algorithm 1. It traverses each record to get the value of key_X from $r[X]$. It accesses $\mathcal{O}_X^{\mathsf{KL}}$ and stores key_X in the ORAM when this key_X has not been stored before. The injective mapping between key_X (also $r[X]$) and $label_X$ is constructed with the incremental variable $card_X$: each time a new key_X appears, $card_X$ is added with 1 to guarantee a unique $label_X$. Finally, the value of $card_X$ is equivalent to the number of distinct key_X (i.e., $|\pi_X|$) and ($\mathcal{O}_X^{\mathsf{KL}}$,$\mathcal{O}_X^{\mathsf{IL}}$) preserves π_X.

Algorithm 2 is proposed for calculating $|\pi_X|$ when a given attribute set X consists of multiple attributes (i.e., $|X| \geq 2$). Now it does not need to input the value from $r[X]$, whose length can be very large as X includes multiple attributes. The injective mapping between $r[X]$ and short key_X solves this challenge well. Recall Property 1 in Section IV-A guarantees:

- There exist two distinct attribute sets $X_1, X_2 \subsetneq X$ such that $X_1 \cup X_2 = X$.

- Before calculating $|\pi_X|$, the ORAMs for X_1 and X_2 have been established denoted by $(\mathcal{O}^{\mathsf{KL}}_{X_1}, \mathcal{O}^{\mathsf{IL}}_{X_1})$ and $(\mathcal{O}^{\mathsf{KL}}_{X_2}, \mathcal{O}^{\mathsf{IL}}_{X_2})$, respectively.

So we can apply the established ORAMs to extract $label_{X_1}$ and $label_{X_2}$, which are uniquely mapped by $r[X_1]$ and $r[X_2]$. As $(r[X_1], r[X_2])$ also maps a unique $r[X]$, line 6 in Algorithm 2 constructs a key_X uniquely corresponding to $r[X]$. Then we can establish the ORAMs for X similar to that for a single attribute. We remark that although we use $r[ID]$ in the algorithms, without loss of generality, $r[ID]$ can be some predefined numbers like row numbers, so $\mathcal{C}$ can directly adopt i as $r_i[ID]$ and do not need to get it by retrieving its ciphertext from $\mathcal{S}$.

c) Analysis: The algorithms in this section allow insertion queries to DB because it works by traversing records one by one. So the newly inserted records can be regarded as untraversed records to continue the process. It does not work with deletion because multiple records can correspond to the same pair stored in $\mathcal{O}^{\mathsf{KL}}_X$, making the deletion incorrect. For example, given $(r_1[X], r_2[X]) = (Alice, Alice)$, and suppose the corresponding pair in $\mathcal{O}^{\mathsf{KL}}_X$ is $(key_X, label_X) = (Alice, 13)$. If $\mathcal{C}$ directly removes $(Alice, 13)$ from $\mathcal{O}^{\mathsf{KL}}_X$ when deleting r_1, then the calculation is incorrect because the pair should be preserved for r_2.

We briefly mention some details about checking if $|\pi_X| = |\pi_{X\cup Y}|$ for any two attribute sets X and Y. These explain how the algorithms here are combined with the set-level task. After calculating $(|\pi_X|, |\pi_{X\cup Y}|)$ with the algorithms, their ciphertexts are stored in $\mathcal{S}$. So $\mathcal{S}$ transfers them to $\mathcal{C}$ and lets $\mathcal{C}$ tell it if the plaintexts are equal. This guarantees that $\mathcal{S}$ learns nothing about the values of $(|\pi_X|, |\pi_{X\cup Y}|)$ besides if they are equal.

D. Oblivious Algorithm based on Oblivious Sorting

In this section, we introduce the second oblivious algorithm for calculating partitions which is based on oblivious sorting. Compared with the ORAM-based algorithm, it is easy to implement. Especially, it allows high parallelism, making it much more efficient and practical in reality.

a) Setup: We still follow the design of $(key_X, label_X)$ introduced in Section IV-B. For each record r and attribute set X, we assign a unique key_X and $label_X$ for $r[X]$.

- $|X| = 1$: If X includes only a single attribute, then key_X is assigned with the value of $r[X]$.
- $|X| \geq 2$: If X consists of multiple attributes, according to the Property 1, we can find two distinct sets $X_1, X_2 \subsetneq X$ such that $X_1 \cup X_2 = X$ and $(label_{X_1}, label_{X_2})$ have been calculated. Then we assign

$$key_X := label_{X_1} \cdot n + label_{X_2}$$

where $label_{X_1}, label_{X_2}$ are two integers in $[n]$. We will show how to extract $label_{X_1}$ and $label_{X_2}$ in the next *Calculation* subroutine.

Algorithm 3: Calculate $|\pi_X|$ with sorting

Input: $A := \{(key^i_X, r_i[ID])\}^n_{i=1}$
Output: Integer $|\pi_X|$, Partition π_X

```
   /* Sort A according to key_X */
1  A' = ObliviousSort(key_X, A)
2  tmp = A'[1][key_X],  card_X = 0
3  for i = 1; i ≤ n; i++ do
4      flag = 1{A'[i][key_X] ≠ tmp}
5      tmp = A'[i][key_X] · flag + tmp · (1 − flag)
6      card_X = card_X + flag
7      A'[i][key_X] = card_X
8  end
   /* Sort A' according to r[ID] */
9  B := ObliviousSort(r[ID], A')
10 Return |π_X| := card_X + 1, π_X := B
```

b) Calculation: After *Setup* subroutine, now we have a column of pairs denoted by $A := \{(key^i_X, r_i[ID])\}^n_{i=1}$ as inputs. Then we show the detailed algorithm in Algorithm 3:

1) We conduct oblivious sorting on the pairs such that they are ordered by key_X. In this way, records with the same key_X are placed consecutively.
2) We traverse the ordered pairs. For records with the same key_X, we assign them a unique integer with the variable $card_X$. We replace key_X with the integer to compress key_X to an integer in $[n]$.
3) Finally, we sort records to make them ordered by $r[ID]$. The final array denoted by B preserves the information about π_X and $(card_X + 1)$ records the number of distinct key_X, e.g., $|\pi_X|$.

The step 2) compresses each distinct key_X to a unique integer in $[n]$. We use and denote them as the unique $label_X$ for key_X. Therefore, although we use key_X for sorting (in Line 1 of Algorithm 3), our final results B only preserve the short $label_X$.

Now we can introduce given two attribute sets X_1, X_2 whose partitions have been calculated, how to extract $label_{X_1}$ and $label_{X_2}$ for each record. Suppose the arrays that preserve (π_{X_1}, π_{X_2}) are (B_{X_1}, B_{X_2}). Then each array is ordered by $r[ID]$ (required by Line 9 in Algorithm 3). So $B_{X_1}[i]$ and $B_{X_2}[i]$ represents the same record r_i as they share the same $r[ID]$. So we just extract

$$label^i_{X_1} := B_{X_1}[i][label_{X_1}],$$

$$label^i_{X_2} := B_{X_2}[i][label_{X_2}]$$

where $(label^i_{X_1}, label^i_{X_2})$ is for record r_i.

c) Analysis: The algorithm applies *bitonic sorting*, which enables $n/2$ parallelism degree for sorting n elements. Also, it is easy to see the algorithm in this section is very simple and requires only $O(1)$ memory in $\mathcal{C}$ (for comparison). Therefore, this algorithm is very suitable for those encrypted databases based on secure enclaves, which require simple programs and small trusted memories. We conduct this algorithm with secure enclaves to show this in experiments (cf. Section VII-D).

V. Extension on Dynamic Databases

In this section, we extend the ORAM-based algorithms in Section IV-C to dynamic databases with both insertions and deletions[1]. It can be applied to construct *the first non-trivial* secure FD discovery protocol on dynamic databases.

A. Dynamic Databases and Trivial Solutions

Dynamic databases are common in real-world scenarios and encrypted databases[6], [22]. $\mathcal{C}$ can insert some new records into DB, making prior FDs invalid. Also, it can delete some records from DB to result in new FDs [52]. Therefore, it is necessary to design secure FD discovery protocols for dynamic databases.

A naive solution is to treat the database after insertions and deletions as an entirely new database to re-conduct the prior two protocols for static databases in Section IV. But this is very expensive. Especially, when there are only a few changes in the database like several records are deleted, it is unacceptable to conduct FD discovery protocols on the whole database. We call such expensive solutions as *trivial*.

Definition 5 (Trivial)**.** *Suppose a database* DB *with n records whose FDs have been discovered. Given any FD $F: X \rightarrow Y$ where X and Y separately include only one single attribute in* DB*, it is trivial to re-validate this FD with $\Omega(n)$ computational complexity after one insertion or deletion.*

We do not count the attribute number in complexity, thus we require X and Y to include only one single attribute in the above definition. Clearly, we hope *non-trivial solutions* can operate only on the newly inserted or deleted records instead of all records to re-validate the FDs.

B. The Extended Setup

We first extend the setup of ORAM-based algorithms in Section IV-C. Specifically, we require the ORAMs to store more information about the database e.g., the frequency of plaintexts.

[1]The update can be regarded as the composition of deletion and insertion.

Algorithm 4: Extended version of Algorithm 1

```
   Input: {(r_i[X], r_i[id])}_{i=1}^n
   Output: Integer |π_X|, ORAMs (O_X^KLF, O_X^IKL)

   /* Initialize ORAMs                                  */
 1 O_X^KLF ← Setup(1^λ), O_X^IKL ← Setup(1^λ)
 2 card_X = 0

   /* Establish ORAMs                                   */
 3 for i = 1; i ≤ n; i++ do
 4     key_X = r_i[X]
 5     ((label_X, fre_X), O_X^KLF) ← Read(key_X, O_X^KLF)
        flag = 1{(label_X, fre_X) ≠ (⊥, ⊥)}

       /* Treat ⊥ as 0 in computation                   */
 6     label_X = flag · label_X + (1 − flag) · card_X
 7     fre_X = fre_X + 1

 8     O_X^IKL ← Write((r_i[ID], (key_X, label_X)), O_X^IKL)
 9     O_X^KLF ← Write((key_X, (label_X, fre_X)), O_X^KLF)
10     card_X = card_X + (1 − flag)
11 end

12 Return |π_X| := card_X, π_X := (O_X^KLF, O_X^IKL)
```

We still follow the design of $(key_X, label_X)$: for each record r and any attribute set X, we map $r[X]$ to a unique $key_X \in [n^2+n]$ and a unique $label_X \in [n]$ where n is the number of rows in DB. Besides, we also establish two ORAMs:

- **Key-(Label, Frequency) ORAM $\mathcal{O}_X^{\mathsf{KLF}}$.** This ORAM is designed to count the number of distinct $r[X]$ and their frequencies. For each distinct $r[X]$, denote its frequency under this attribute as fre_X, this ORAM stores the pair $(key_X, (label_X, fre_X))$.
- **ID-(Key, Label) ORAM $\mathcal{O}_X^{\mathsf{IKL}}$.** For each record r, this ORAM stores a pair $(r[ID], (key_X, label_X))$ where $r[ID]$ is the unique representation of record r.

For ease of presentation, we summarize the modification of ORAMs from the original algorithms to the extended version as below:

$$\mathcal{O}_X^{\mathsf{KL}} : (key_X, label_X) \rightarrow \mathcal{O}_X^{\mathsf{KLF}} : (key_X, (label_X, fre_X)),$$

$$\mathcal{O}_X^{\mathsf{IL}} : (r[ID], label_X) \rightarrow \mathcal{O}_X^{\mathsf{IKL}} : (r[ID], (key_X, label_X)).$$

where fre_X denotes the frequency of the corresponding $r[X]$ under the attribute set X. The modified ORAMs additionally store fre_X and key_X. The length of fre_X is no more than $\lceil \log n \rceil$. The length of key_X is no more than $2\lceil \log n \rceil + 1$ when $|X| \geq 2$ (cf. Section IV-B).

C. The Extended Calculation

Now we propose new oblivious algorithms for calculating partitions on dynamic databases. For any attribute set X with $|X| = 1$, we show the algorithm for calculating $(\pi_X, |\pi_X|)$ in Algorithm 4. It extends Algorithm 1 to process the modified ORAMs and the frequency information about $r[X]$. We do not show the algorithm for the case where $|X| \geq 2$ for brevity. It is identical to Algorithm 4 in most steps but it needs to get key_X from other ORAMs, which has been fully shown in Algorithm 2 (line 4-6).

a) Insertion: The ORAM-based methods inherently support insertions because they always calculate partitions by traversing records one by one. So the inserted records can be treated as untraversed records. We can directly continue Algorithm 4 on these records to update π_X and $|\pi_X|$.

b) Deletion: The deletion can be completed naturally with the modified ORAMs. The detailed algorithm is shown in Algorithm 5. For a deleted record r and any attribute set X, we conduct two steps to delete its information in $(\pi_X, |\pi_X|)$:

1) We apply its ID $r[ID]$ to delete its pair in $\mathcal{O}_X^{\mathsf{IKL}}$. In this step, we can also find its corresponding key_X.
2) We apply its key_X to process its pair in $\mathcal{O}_X^{\mathsf{KLF}}$. If the frequency of $r[X]$ (i.e., fre_X) is larger than 1, then this pair is also shared by other records. So we subtract fre_X with 1. Otherwise, this pair belongs to only the deleted record and we remove it.

We note the deletion for any two distinct attribute sets X and Y can be done in parallel. However, the partition calculation and insertion process for two distinct attribute sets must follow the order determined by the database-level task because they need the guarantee Property 1 (cf. Section IV-A).

The ideal security goal in this paper does not require insertion and deletion to be indistinguishable for $\mathcal{S}$ as it allows the leakage of database size. But the extended ORAM-based method has the potential to achieve indistinguishability between insertion and deletion. This can be done by treating deletion as insertion but now (1) we subtract fre_X with 1 while insertion adds fre_X with 1; (2) we remove the pair in ORAMs while insertion stores a new pair in ORAMs.

Algorithm 5: Re-calculate $(\pi_X, |\pi_X|)$ after deletion

Input: Integer $card_X$, The ID of deleted record denoted by $r[ID]$
Output: Integer $|\pi_X|$, ORAMs $(\mathcal{O}_X^{\mathsf{KLF}}, \mathcal{O}_X^{\mathsf{IKL}})$

```
1  ((key_X, label_X), O_X^IKL) ← Read(r[ID], O_X^IKL)
2  ((label_X, fre_X), O_X^KLF) ← Read(key_X, O_X^KLF)
   /* Delete key_X and label_X iff fre_X = 1 */
3  flag = 1{fre_X ≠ 1}
   /* Define ⊥ + 0 = ⊥ */
4  key_X = flag · key_X + (1 − flag) · ⊥
5  label_X = flag · label_X + (1 − flag) · ⊥
6  fre_X = flag · (fre_X − 1) + (1 − flag) · ⊥
7  card_X = card_X − (1 − flag)
8  O_X^KLF ← Write((key_X, (label_X, fre_X)), O_X^KLF)
9  O_X^IKL ← Write((r[ID], (⊥, ⊥)), O_X^IKL)
10 Return |π_X| := card_X, π_X := (O_X^KLF, O_X^IKL)
```

VI. Security Analysis

In this section, we prove the protocols achieve secure FD discovery if they consist of: (1) the database-level top-down method; (2) the set-level checking; (3) one of our algorithms for calculating partitions in the attribute level. That means when applying them on a database DB, they leak nothing besides

$$\mathcal{L}(\mathsf{DB}) := \{\mathsf{Size}(\mathsf{DB}), \mathsf{FD}(\mathsf{DB})\}$$

where $\mathsf{Size}(\mathsf{DB})$ is the database size (i.e., the number of rows and columns) and $\mathsf{FD}(\mathsf{DB})$ is the set of all FDs in DB. Recall the security model in Section III-B, we consider a PPT adversary $\mathcal{A}$ that tries to distinguish the real world (**Real**) and ideal world (**Ideal**). In **Real**, we conduct the protocol denoted by Π on the encrypted database $\widehat{\mathsf{DB}}$ to discover FDs. In **Ideal**, a simulator Sim simulates execution of Π on $\widehat{\mathsf{DB}}$ in the real world based on the information $\mathcal{L}(\mathsf{DB})$. We assume Π always adopts simple checking in the set level and top-down method in the database level. Then we define the advantage

$$\mathsf{Adv}^{\Pi}_{\mathsf{Sim},\mathcal{L}}(\mathcal{A}) := |\Pr[\mathbf{Real}^{\Pi}_{\mathcal{A}} = 1] - \Pr[\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathsf{Sim},\mathcal{L}} = 1]|$$

Theorem 2 (Static databases)**.** *The protocol* Π *achieves secure FD discovery on any static database* DB *if it adopts the algorithm in Section IV-C or Section IV-D in the attribute level.*

Proof. (Sketch) We give a series of game transitions to show how to construct the simulator Sim. Denote the experiment in **Real** as G^0 where $\mathcal{C}$ and $\mathcal{S}$ executes Π interactively. We note, in the database level, G^0 inherently applies only $\mathcal{L}(DB)$ to perform the task. So Sim can simulate this level identically with $\mathcal{L}(DB)$.

We give game G^1 that is the same as G^0 but in the set level, for any two attribute sets X and Y, it preserves two random strings instead of the ciphertexts of $(|\pi_X|, |\pi_{X\cup Y}|)$. It accesses the two strings (to simulate accessing the ciphertexts) but determines if $X \to Y$ holds according to $\mathcal{L}(DB)$.

Denote the advantage of any PPT adversary $\mathcal{A}_0$ distinguishing G^0 and G^1 as $\mathsf{Adv}(\mathcal{A}_0)$, then it is negligible [25].

Define game G^2 that is the same as G^1 but in the attribute level, it conducts the partition calculation on a database DB' generated by Sim. The size of DB' is $\mathsf{Size}(DB)$. Define the advantage of any PPT adversary $\mathcal{A}_1$ distinguishing G^1 and G^2 as $\mathsf{Adv}(\mathcal{A}_1)$. Now we show $\mathsf{Adv}(\mathcal{A}_1)$ is negligible in λ.

For the algorithm based on oblivious sorting, it has fixed access patterns in $\mathcal{S}$ [4]. Thus $\mathsf{Adv}(\mathcal{A}_1)$ is equal to that of any PPT adversary $\mathcal{A}_2$ distinguishing semantically secure ciphertexts of two databases with the same size denoted by $\mathsf{Adv}(\mathcal{A}_2)$ which is negligible in λ. For the algorithm based on ORAM, besides fixed access patterns on some variable ciphertexts, it applies ORAM to access a sequence with a fixed length. So $\mathsf{Adv}(\mathcal{A}_1)$ is equal to $\mathsf{Adv}(\mathcal{A}_2) + \mathsf{Adv}(\mathcal{A}_3)$ where $\mathsf{Adv}(\mathcal{A}_3)$ denotes the advantage of any PPT adversary $\mathcal{A}_3$ distinguishing two sequences with the same length accessed by ORAM. Note $\mathsf{Adv}(\mathcal{A}_3)$ is also negligible in λ [55], so when the experiment in $\mathbf{Ideal}$ is identical to G^2, we have

$$\mathsf{Adv}^{\Pi}_{\mathsf{Sim},\mathcal{L}}(\mathcal{A}) = \mathsf{Adv}(\mathcal{A}_0) + \mathsf{Adv}(\mathcal{A}_1) \leq \mathsf{negl}(\lambda).$$

□

Theorem 3 (Dynamic databases)**.** *The protocol* Π *achieves secure FD discovery on databases with insertions/deletions* DB *if it adopts the algorithm in Section V in the attribute level.*

Proof. (Sketch) We only prove the basic case where there is only one insertion/deletion. Multiple insertions and deletions can be considered by repeating the basic case.

We still use the game transition. Define G^0 to be identical to the experiment in $\mathbf{Real}$. Where there is one insertion/deletion, the partitions will be re-calculated in the attribute level, and the tasks in set and database levels will be re-executed.

We still define G^1 to be the same as G^0 besides the task in the set level is done with random strings according to the updated $\mathcal{L}(\mathsf{DB})$ after insertion/deletion. Define game G^2 to be the same as G^0 besides the database and insert/delete are generated by Sim. Then, similarly, the algorithm for insertion/deletion also consists of fixed access patterns towards variable ciphertexts and using ORAM to access a sequence with a fixed length. Therefore, denote the advantage of any adversary $\mathcal{A}_0$ distinguishing G^0 and G^1 as $\mathsf{Adv}(\mathcal{A}_0)$ and the advantage of any adversary $\mathcal{A}_1$ distinguishing G^1 and G^2 as $\mathsf{Adv}(\mathcal{A}_1)$. When the experiment in $\mathbf{Ideal}$ is identical to G^2, we have

$$\mathsf{Adv}^{\Pi}_{\mathsf{Sim},\mathcal{L}}(\mathcal{A}) = \mathsf{Adv}(\mathcal{A}_0) + \mathsf{Adv}(\mathcal{A}_1) \leq \mathsf{negl}(\lambda).$$

□

VII. Evaluation

In this section, we evaluate our methods for obliviously calculating partitions including the original ORAM-based method (Or-ORAM), the extended ORAM-based method (Ex-ORAM), and the method applying oblivious sorting (Sort). The whole FD discovery protocol consists of repeated partition calculations for different attribute sets.

A. Setup and Dataset

There are three important metrics considered when evaluating our methods: (1) runtime of the algorithms; (2) storage usage in $\mathcal{S}$; (3) memory usage in $\mathcal{C}$. These metrics are commonly considered in encrypted databases [13], [44], [45] for practicality, thus reflect the actual performance of our methods in encrypted databases. In the experiments, we will demonstrate the following claims with the three metrics.

Obliviousness: *The obliviousness guarantees our methods perform identically for* $\mathcal{S}$ *on datasets with different distributions.*

Scalability: *The ORAM-based methods perform better in scalability because of their lower calculation complexity.*

Practicality: Sort *is the most practical since it is simple and has the potential for parallelism.*

Flexibility: Ex-ORAM *enables fast insertion and deletion and thus has great flexibility.*

All methods are implemented as a modular client-server application in Python 3.10. The client $\mathcal{C}$ and server $\mathcal{S}$ are separately played by an Ubuntu 22.04.3 machine with Intel Xeon Platinum 8369B CPU (16 cores, 2.70GHz), 64GB of memory, and an 80GB hard disk. The bandwidth is 1 Gbps. Note $\mathcal{C}$ will apply only a little resource to be consistent with our assumptions. We adopt the AES/CBC encryption to guarantee semantically secure encryption. The key length is 128 bits. For PathORAM settings, we follow [8], [10] to place $Z = 4$ blocks in each bucket. We limit the stash in $\mathcal{C}$ to store at most $7 \log n$ blocks. For some experiments in Section VII-D which apply SGX, we set up SGX on an Alibaba Cloud Linux release 3 with an Intel Xeon Platinum 8369B CPU (4-core, 2.70GHz), 16GB memory (including 8GB secure memory), and 80GB hard disk and SGX v2.19. The codes are going to be open-sourced soon.

We adopt four datasets in our experiments including one synthetic dataset and three real-world datasets with different distributions. The synthetic dataset named *RND* can be generated with arbitrary columns and rows and each plaintext in the cell level is randomly selected from $[1, 2^{20}]$. The three real-world datasets are also used in prior work [41] about FD discovery and we summarize them in Table I. The *Adult* originates from census data and *Letter* includes the information about English alphabet. The *Flight* comprises flight route data, which are extracted from data streams.

Dataset	# Columns	# Rows	# Size
Adult	14	48,842	3528KB
Letter	16	20,000	695KB
Flight	20	500,000	71MB

TABLE I. The summary of datasets

B. Obliviousness

As the most important property, obliviousness guarantees that our methods perform identically in the view of $\mathcal{S}$ even on datasets with different distributions. That means *the storage usage* in $\mathcal{S}$ and *runtime* of the whole method are identically distributed to all datasets with the same size. To show this, we conduct all the methods on the four datasets and compare the storage usage and runtime. We randomly selected 2^{13} rows from each dataset to ensure the same size. We test the partition calculation for attribute set X for two cases where $|X| = 1$ and $|X| \geq 2$. Our methods also perform identically on two attribute sets X and Y when $|X|, |Y| > 2$ (cf. Section IV-B), thus we consider $|X| \geq 2$ in the whole. There are a total of four groups of experiments we conducted:

1) S_1 : For each real-world dataset, we randomly picked 9 single columns in it to conduct partition calculation with our methods and record the runtime as set S_1;
2) S_2 : For each real-world dataset and any integer i in $[2, 10]$, we randomly picked an attribute set X such that $|X| = 1$. We conduct our methods on X to calculate partitions and record runtime as set S_2;
3) S_3 : We run our methods on the same single column of *RND* for 9 times and record the runtime as set S_3;
4) S_4 : We run our methods with the the same attribute X in *RND* for 9 times where $|X| = 2$ and record runtime as set S_4.

The obliviousness should guarantee that (S_1, S_3) follow the same distribution and (S_2, S_4) follow the same distribution. To verify this, we adopt the two-sample the two-sample Kolmogorov–Smirnov (KS) test [38] on (S_1, S_3) and (S_2, S_4), respectively. The test outputs p-values that indicate if we have significant evidence to claim they follow distinct distributions. We show the p-values in Table II. Generally, we can claim the samples follow distinct distributions only when the p-value is very small (< 0.05). However, all p-values in Table II are no smaller than 0.35, which shows we have no significant evidence to claim they follow distinct distributions. We also show the average storage usage in $\mathcal{S}$ in the last column of Table II. The storage usage is always nearly identical when we conduct the methods on different datasets. Therefore, in the remaining experiments, we apply only *RND* dataset to show the performance of our methods on runtime and storage cost in $\mathcal{S}$.

Methods	Case	Adult	Letter	Flight	Sto (MB)
Or-ORAM	$\lvert X\rvert = 1$	0.35	0.73	0.35	30.97
	$\lvert X\rvert \geq 2$	0.73	0.73	0.35	31.01
Ex-ORAM	$\lvert X\rvert = 1$	0.98	0.98	0.35	39.12
	$\lvert X\rvert \geq 2$	0.35	0.35	0.98	39.11
Sort	$\lvert X\rvert = 1$	0.98	0.73	0.98	0.25
	$\lvert X\rvert \geq 2$	0.60	0.35	0.35	0.25

TABLE II. The two-sample KS test p-value on the runtime of methods on different datasets. **Sto** represents the storage usage in $\mathcal{S}$.

Methods	Computation	Storage in $\mathcal{S}$
ORAM	$O(n \log n(1 + \log^2 \log n))$	$O(n)$
Sort	$O(n \log^2 n)$	$O(n)$

TABLE III. Summary of methods. This follows [10] but additionally assumes all sorting are done by bitonic sorting [4].

C. Scalability

Here we evaluate the *row sclability* of our methods, i.e., the performance on the dataset with different numbers of rows. Suppose there are n rows in the dataset and we calculate the partition under any attribute set X with our methods. We show the theoretical complexity in Table III and experimental results in Fig. 4 and Fig. 5.

a) Computation: For ORAM-based methods including Or-ORAM and Ex-ORAM, their computation complexity is smaller than that of Sort, thus they require less time when n increases. The runtime is shown in Fig. 4.

- For $|X| = 1$, Sort is much more expensive than ORAM-based methods when $n > 2^{11}$.
- For $|X| \geq 2$, Sort is still much more expensive than Or-ORAM when $n > 2^{11}$, but is comparative to Ex-ORAM when $n < 2^{14}$.

From $|X| = 1$ to $|X| \geq 2$, ORAM-based methods require much more additional time. This is because they have to additionally access the ORAMs of X's two subsets, which is very costly. Ex-ORAM is more expensive than Or-ORAM since it needs to store and access more information in ORAMs.

b) Storage: For each method, the storage cost in $\mathcal{S}$ depends on only n. We show the storage cost in Fig. 5. All three methods require $O(n)$ storage for storing π_X for any attribute set X. The performance of Sort is the best because it stores only the ciphertexts of $label_X$ (The column of $r[ID]$ is stored only once). ORAM-based methods require much more storage for two reasons. Firstly, it requires additional dummy ciphertexts to hide real ciphertexts. Secondly, it needs to store more information like key_X and $r[ID]$ in the ORAMs for each attribute set X. This also results in that

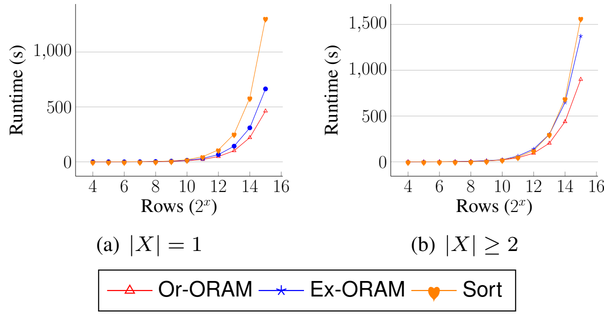


Fig. 4. Row scalability for runtime

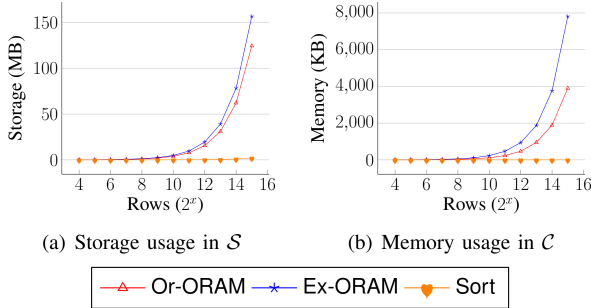
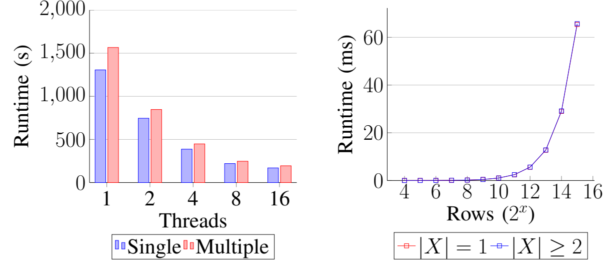


Fig. 5. Row scalability for storage usage in $\mathcal{S}$ and memory usage in $\mathcal{C}$. They are the same for both $|X| = 1$ and $|X| \geq 2$.

Ex-ORAM uses more storage than Or-ORAM as Ex-ORAM additionally stores frequency information about plaintexts.

c) Memory: We also evaluate the memory usage in $\mathcal{C}$. We show the required memory in $\mathcal{C}$ to maintain information for calculating partitions in Fig. 5. Sort performs best: it needs to store only the 128-bit secret key for encryption/decryption. It proceeds by retrieving two ciphertexts, processing them, and sending back them. The whole process in $\mathcal{C}$ also requires only 56.2MB memory (mainly from the socket functions) no matter how large n is. ORAM-based methods cost $O(n)$ memory (like [5], [10]) because the non-recursive PathORAM requires storing *stash* and *position map*. The storage requirement can be reduced by adopting more advanced ORAMs [3], [43] at the cost of runtime. Here this storage is worthwhile as it is much smaller compared with storing $r[X]$ when $|X|$ is large. We note this storage is constant due to our design of $(key_X, label_X)$ for any attribute set X. It is shown at most 0.8MB is needed when $n = 2^{15}$. Therefore, for any ORAM, $\mathcal{C}$ can also transfer the corresponding stash and position map to $\mathcal{S}$ encrypted and retrieve them quickly when using the ORAM.

Therefore, ORAM-based methods perform better in runtime, and Sort performs much better in both server-side storage and client-side memory. Ex-ORAM should not be chosen if there is no deletion as Or-ORAM is faster and uses smaller storage and memory.



(a) The performance with multiple threads.

(b) The runtime in SGX.

Fig. 6. Performance of Sort in parallelism and SGX.

D. Practicality

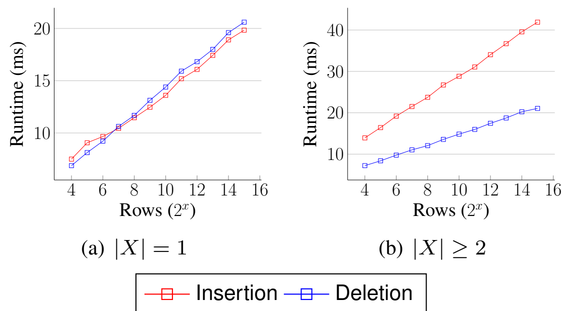
In this section, we show Sort is the most practical method among the three methods. It allows a high parallelism degree (at most $n/2$), which makes it can be more efficient in reality. Moreover, it is easy to implement, for example, in secure enclaves. The simplicity enables it to be easily integrated with all existing encrypted databases that apply secure enclaves.

To show the parallelism, we run Sort on *RND* with 2^{15} rows under different numbers of threads. Each thread is responsible for a part of Sort. The experimental results are shown in Fig. 6(a). It shows that one thread needs over 1500 seconds to complete the whole Sort. But with 16 threads, the same task can be finished within 200 seconds, which is a huge improvement. The effectiveness of adding threads also decreases with more and more number threads added. In the beginning, adding threads from 1 to 2 reduces half of the runtime. In the end, adding threads from 8 to 16 affects the runtime very incrementally.

Sort is easy to deploy, thus we also deploy it in secure enclaves to further understand its efficiency. Note the secure enclave can store the plaintexts in secure memory which $\mathcal{S}$ cannot see, so the expensive data transfer between $\mathcal{C}$ and $\mathcal{S}$ and most re-encryption can be discarded. This advantage significantly accelerates Sort, which is shown in Fig. 6(b). With SGX, the runtime for $|X| = 1$ and $|X| \geq 2$ are nearly identical so we can see the two curves are overlapped. The efficiency improvement is huge: without SGX, the runtime is over 1500s to process the case of $|X| \geq 2$ when $n = 2^{15}$, but now we only need 66 ms ($22,000\times$ speedup) for this.

E. Flexibility

Here we test the performance of Ex-ORAM on insertion and deletion operations. We generate the dataset *RND* with rows from 2^4 to 2^{15}. We insert all these rows first and then delete all of them. We count the average runtime per insertion/deletion and show the results in Fig. 7. Both the insertion and deletion time increases with n, the corresponding computational complexity is $O(\log n \log^2 \log n)$. The time increase results from accessing ORAM with a larger size. Fortunately, even if $n = 2^{15}$, the insertion and deletion under



(a) $|X| = 1$ (b) $|X| \geq 2$

Fig. 7. Insertion and deletion efficiency

$|X| = 1$ are completed within only 22ms. The insertion under $|X| \geq 2$ is more expensive but is still done within 43 ms. And the deletion under $|X| \geq 2$ is more efficient, which is executed within 22ms.

When $|X| = 1$, both insertion and deletion access two ORAMs, resulting in a similar efficiency in Figure 7(a). The insertion is a little more expensive in the beginning because the first insertion requires the setup of ORAMs. It becomes faster with n increasing because there are additional $O(n)$ decryption for deletion. For example, before the first insertion, $\mathcal{S}$ knows the ORAM stores no item, thus $\mathcal{C}$ does not need to decrypt ciphertexts retrieved from $\mathcal{S}$ in the first insertion. When $|X| = 2$, the insertion time is twice as much as the deletion time. This is because the insertion accesses four ORAMs while deletion accesses only two ORAMs.

VIII. Related work

a) FD discovery: As a critical task in databases, FD discovery has been studied for several decades [16], [23], [32], [52], [60]. Most of them focus on improving its efficiency or extending the cover range like *conditional FD* without security. Besides the partition-based methods, there are also some methods that use difference-sets [59] and agree-sets [34]. However, the partition-based method is more friendly to be designed obliviously. It is still open how the other FD methods can be reconstructed to be oblivious.

The security problem of FD discovery is initiated by two works [14], [17]. Ge et al. [17] study how to achieve FD discovery with multiple data owners without leaking their privacy to each other (i.e., the multi-party computation). Dong et al. [14] are the first to discuss FD discovery in encrypted databases. However, their construction provides only a very weak security guarantee: partial frequency information about plaintexts is allowed to be revealed to $\mathcal{S}$. As the frequency leakage has been recognized as dangerous to $\mathcal{C}$, it may be infeasible to deploy such a construction in reality. This paper follows the work of Dong et al. and is the first to define and achieve truly secure FD discovery with minimal leakage, providing a very strict security guarantee.

b) Obliviousness: In the last decade, obliviousness has become one of the most important topics in encrypted databases because extensive works [24], [26], [28] have shown how access patterns can leak dangerous and sensitive information. Now achieving oblivious query processing such as join and range query has motivated a lot of work in database community [10], [27], [54]. However, all of them do not consider database maintenance like FD discovery. So this paper provides the first solution to all these encrypted databases for FD discovery. It is easy to integrate our work with them while still keeping a formal and strict security guarantee. There are also extensive works [2], [3], [33], [43], [49], [51], [55], [56] about improving the efficiency of the primitives including ORAM and oblivious sorting. As our work applies the two primitives in a black-box style, any optimization can be applied easily for a more efficient FD discovery. Notably, the results in [27], [13] show the costs can be much reduced when deploying the primitives in secure enclaves or systems with high parallelism.

IX. Conclusion

In this paper, we first define secure FD discovery in encryption databases with minimal leakages which consist of the database size and discovered FDs. Then we propose two specific protocols to achieve secure FD discovery. The two protocols adopt Oblivious RAM and oblivious sorting. They can be integrated with most existing encrypted databases to achieve FD discovery with a strict security guarantee. They are the first step to address database maintenance in encrypted databases. In the future, we will further improve the efficiency of our protocols and achieve more tasks about database maintenance in encrypted databases.

References

[1] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *Symposium on Simplicity in Algorithms*, pages 8–14. SIAM, 2020.

[2] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: optimal oblivious ram. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30*, pages 403–432. Springer, 2020.

[3] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Optimal oblivious parallel ram. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2459–2521. SIAM, 2022.

[4] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[5] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. psolute: Efficiently querying databases while providing differential privacy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2262–2276, New York, NY, USA, 2021. Association for Computing Machinery.

[6] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. 2014.

[7] TH Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2201–2220. SIAM, 2018.

[8] Zhao Chang, Dong Xie, and Feifei Li. Oblivious ram: A dissection and experimental evaluation. *Proceedings of the VLDB Endowment*, 9(12):1113–1124, 2016.

[9] Zhao Chang, Dong Xie, Feifei Li, Jeff M. Phillips, and Rajeev Balasubramonian. Efficient and oblivious query processing for range and knn queries (extended abstract). In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1487–1488, 2022.

[10] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. Towards practical oblivious join. In *Proceedings of the 2022 International Conference on Management of Data*, pages 803–817, 2022.

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

[12] Laura Chiticariu, Mauricio A Hernández, Phokion G Kolaitis, and Lucian Popa. Semi-automatic schema integration in clio. In *VLDB*, volume 7, pages 1326–1329, 2007.

[13] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 655–671, 2021.

[14] Boxiang Dong and Wendy Wang. Frequency-hiding dependency-preserving encryption for outsourced databases. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 721–732, 2017.

[15] F Betül Durak, Thomas M DuBuisson, and David Cash. What else is revealed by order-revealing encryption? In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1155–1166, 2016.

[16] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):683–698, 2010.

[17] Chang Ge, Ihab F Ilyas, and Florian Kerschbaum. Secure multi-party functional dependency discovery. *Proceedings of the VLDB Endowment*, 13(2):184–196, 2019.

[18] Michael T Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o (n log n) time. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 684–693, 2014.

[19] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2451–2468. USENIX Association, August 2020.

[20] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE symposium on security and privacy (SP)*, pages 655–672. IEEE, 2017.

[21] Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwar, Shantanu Sharma, and Sumaya Almanee. Obscure: Information-theoretic oblivious and verifiable aggregation queries. *Proceedings of the VLDB Endowment*, 12(9):1030–1043, 2019.

[22] Kun He, Jing Chen, Qinxi Zhou, Ruiying Du, and Yang Xiang. Secure dynamic searchable symmetric encryption with constant client storage cost. *IEEE Transactions on Information Forensics and Security*, 16:1538–1549, 2020.

[23] Yka Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[24] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Ndss*, volume 20, page 12. Citeseer, 2012.

[25] Jonathan Katz and Yehuda Lindell. Introduction to modern cryptography. *(No Title)*, 2014.

[26] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340, 2016.

[27] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *Proceedings of the VLDB Endowment*, 13(11).

[28] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314. IEEE, 2018.

[29] Dongjie Li, Siyi Lv, Yanyu Huang, Yijing Liu, Tong Li, Zheli Liu, and Liang Guo. Frequency-hiding order-preserving encryption with small client storage. *Proc. VLDB Endow.*, 14(13):3295–3307, sep 2021.

[30] Mingyu Li, Xuyang Zhao, Le Chen, Cheng Tan, Huorong Li, Sheng Wang, Zeyu Mi, Yubin Xia, Feifei Li, and Haibo Chen. Encrypted databases made secure yet maintainable. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 117–133, Boston, MA, July 2023. USENIX Association.

[31] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the n log n barrier for oblivious sorting? In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2419–2438. SIAM, 2019.

[32] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Discover dependencies from data—a review. *IEEE Transactions on Knowledge and Data Engineering*, 24:251–264, 2012.

[33] Zheli Liu, Yanyu Huang, Jin Li, Xiaochun Cheng, and Chao Shen. Divoram: Towards a practical oblivious ram with variable block size. *Information Sciences*, 447:1–11, 2018.

[34] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *International Conference on Extending Database Technology*, pages 350–364. Springer, 2000.

[35] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. Incremental Offline/Online PIR. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1741–1758, Boston, MA, August 2022. USENIX Association.

[36] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296. IEEE, 2018.

[37] Payman Mohassel and Mike Rosulek. Non-interactive secure 2pc in the offline/online and batch settings. Cryptology ePrint Archive, Paper 2017/125, 2017. https://eprint.iacr.org/2017/125.

[38] Marco Monge. Two-sample kolmogorov-smirnov tests as causality tests. a narrative of latin american inflation from 2020 to 2022. *Revista Chilena de Economía y Sociedad*, 2023.

[39] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655, 2015.

[40] Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *International Conference on Database Theory*, pages 189–203. Springer, 2001.

[41] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015.

[42] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*, pages 821–833, 2016.

[43] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious ram with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882. IEEE, 2018.

[44] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. *Cryptology ePrint Archive*, 2016.
[45] Raluca A. Popa, Frank H. Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. *2013 IEEE Symposium on Security and Privacy*, pages 463–477, 2013.
[46] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, pages 85–100, 2011.
[47] Lina Qiu, Georgios Kellaris, Nikos Mamoulis, Kobbi Nissim, and George Kollios. Doquet: Differentially oblivious range and join queries with private data structures. *Proc. VLDB Endow.*, 16(13):4160–4173, 2023.
[48] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 373–384, 2021.
[49] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious {RAM}. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, 2015.
[50] Daniel S Roche, Daniel Apon, Seung Geol Choi, and Arkady Yerukhimovich. Pope: Partial order preserving encoding. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142, 2016.
[51] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks. Cryptology ePrint Archive, Paper 2023/1236, 2023. https://eprint.iacr.org/2023/1236.
[52] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. Dynfd: Functional dependency discovery in dynamic datasets. In *International Conference on Extending Database Technology*, 2019.
[53] Masoumeh Shafieinejad, Suraj Gupta, Jin Yang Liu, Koray Karabina, and Florian Kerschbaum. Equi-joins over encrypted data for series of queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1635–1648, 2022.
[54] Shantanu Sharma, Yin Li, Sharad Mehrotra, Nisha Panwar, Komal Kumari, and Swagnik Roychoudhury. Information-theoretically secure and highly efficient search and row retrieval. *Proceedings of the VLDB Endowment*, 16(10):2391–2403, 2023.
[55] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.
[56] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.
[57] Yannis Vassiliou. Functional dependencies and incomplete information. 1980.
[58] Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, and Siu Ming Yiu. Secure query processing with data interoperability in a cloud database environment. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1395–1406, 2014.
[59] Catharine Wyss, Chris Giannella, and Edward Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In Yahiko Kambayashi, Werner Winiwarter, and Masatoshi Arikawa, editors, *Data Warehousing and Knowledge Discovery*, pages 101–110, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
[60] Renjie Xiao, Yong'an Yuan, Zijing Tan, Shuai Ma, and Wei Wang. Dynamic functional dependency discovery with dynamic hitting set enumeration. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 286–298, 2022.
[61] Hong Yao and Howard J Hamilton. Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16:197–219, 2008.