Pattern-Hiding Encrypted Multi-Maps with Support for Join Queries

Jinguo Li, *Member, IEEE,* Delong Cui, Junqin Huang, Linghe Kong, *Senior Member, IEEE*

***Abstract*—The recently proposed Join Cross-Tags Protocol (JXT) addresses the long-standing issue of excessive query overhead in table joins within Searchable Symmetric Encryption (SSE). As a purely symmetric-key solution, JXT supports efficient conjunctive queries over equi-joins of encrypted tables without requiring any pre-computation during the setup phase. However, JXT has a potential limitation: it may inadvertently reveal the actual volumes of identifiers corresponding to attribute-value pairs, as well as the result values of the join queries. In this paper, we propose JXTMM (JXT multi-map), the first join query scheme designed to hide both volume patterns and result patterns. JXTMM is capable of concealing identifier volumes, preventing the server from learning the actual volumes of attribute-value pairs, and shifting the checkability of join results to the client side, thereby eliminating result pattern leakage. We provide a formal security proof for JXTMM, along with a comprehensive efficiency analysis. Experimental results demonstrate that JXTMM not only performs efficiently on table join queries but also effectively achieves volume-hiding in such queries.**

***Index Terms*—Searchable Encryption, Volume-Hiding, Join Queries, Encrypted Multi-map**

I. Introduction

With the rise of cloud computing and storage, individuals and organizations increasingly store large volumes of data on off-site hosting platforms. However, cloud servers are vulnerable to privacy leakage of hosted data. To address this challenge, several methods for encrypted data search have been proposed. For example, Symmetric Searchable Encryption (SSE), a specific case of Structured Encryption (STE) [1], enables data owners to outsource encrypted data to cloud servers while retaining the ability to retrieve it through keyword searches. To further improve retrieval efficiency, Curtmola et al. [2] proposed an SSE scheme based on inverted indexes. Additionally, Wang et al. [3] enhanced the security of SSE by leveraging the concept of privacy-preserving set intersection to protect retrieval patterns. Despite these advancements, most SSE-based schemes only support single-keyword searches [2]–[6], which limits search expressiveness. Furthermore, single-keyword searches often incur high communication costs in practical applications.

To overcome the above weaknesses, the Oblivious Cross-Tags (OXT) protocol [7] is introduced to enable conjunctive and general Boolean queries on encrypted relational databases. This is accomplished by employing two specialized data structures: the TSet (a cryptographic inverted index) and the XSet (a list of hash pairs). For instance, Cash et al. [8] has achieved asymptotic improvements over the prior OXT scheme by constructing a generic dictionary structure. It can reach optimal leakage minimization, high search computation efficiency, and parallelism search operations. Thus, the scheme can support operations on a dataset with ten of billions of record/keyword pairs. Faber et al. [9] extended search to rich and comprehensive query types by reducing range queries to a disjunction of exact keywords, while other sub-queries (such as substring, wildcard, and phrase queries) are performed with homomorphic computation on encrypted positional information. Patranabis et al. [10] introduced "dynamic cross-tags" to facilitate the dynamic addition and deletion of records in encrypted databases. Nevertheless, the OXT protocol is not efficient in handling table join queries, since it requires an exponentiation operation for every attribute-value pair in the database once the data is encrypted and the XSet is computed. To address this efficiency problem, schemes such as a supporting join queries encryption scheme (SPX) [11], a variant of SPX by introducing the technique of partially pre-computed joins (CNR) [12] are proposed. But they relies heavily on pre-computing all possible joins and storing all the results in the encrypted database.

Thus, the Join Cross-Tags (JXT) protocol [13] is proposed to enable efficient conjunctive queries on equi-joins of encrypted tables. It is a purely symmetric-key solution without any pre-computation at setup. Specifically, JXT decomposes join queries into sub-query on each respective table, and executes these join queries across tables with join attributes. Each table processes its own sub-query independently, and generates two search tokens for each sub-query: one for all entries in this table, and another (termed "cross-marked auxiliary token") for containing entries from the same table and others. Once the two search tokens are used to locate matching entries in the TSet, their membership is verified in the XSet. Following JXT, several improved schemes have been proposed to enhance efficiency, security, and expressiveness in join queries. For instance, TNT-QJ [14] for practical Boolean queries, OTJXT [15] for eliminating the conditional combination pattern leakage, and a dynamic multi-user protocol FBJXT [16] providing forward and backward security.

This work was supported in part by the National Natural Science Foundation of China under Grants 62172276 and 62372285; in part by Yunnan Key Research Program grant 202402AD080004; in part by the Shanghai Action Plan for Science, Technology and Innovation grant 24BC3201300; and in part by Startup Fund for Young Faculty at SJTU (SFYF at SJTU) grant 25X010502613.

J. Li and D. Cui are with the College of Computer Science and Technology, Shanghai University of Electric Power, Shanghai 201306, China (e-mail: lijg@shiep.edu.cn; cdl31900@163.com)

J. Huang and L. Kong are with the School of Computer Science, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: junqin.huang@sjtu.edu.cn; linghe.kong@sjtu.edu.cn; Corresponding author: Junqin Huang)

TABLE I: Comparison with previous schemes.

Scheme	Search Type	Storage Cost	Computation Cost		Communication Cost	Pattern Hiding	
			Client	Server		Volume	Result
XORMM [17]	Single	$1.23n{\cdot}ev+\beta$	$O(\ell_{\max})$	$O(\ell_{\max})$	$O(\ell_{\max})$	✓	×
OXT [7]	Conjunctive	$(xt+ev+y)n$	$O(x\cdot\ell_1)$	$O(\ell_1)+O(x\cdot\ell_1)$	$O(x\cdot\ell_1)$	×	×
JXT [13]	Join	$(ev+y)n+xt{\cdot}m{\cdot}T$	$O(\ell_1+\ell_2)$	$O(\ell_1+\ell_2)+O(\ell_1\ell_2)$	$O(\ell_1+\ell_2)+O(\|DB(q)\|)$	×	×
OTJXT [15]	Join	$(ev+y)n\cdot T$	$O(\ell_1+\ell_2)$	$O(\ell_1+\ell_2)$	$O(\ell_1+\ell_2)+O(\|DB(q)\|)$	×	×
TNT-QJ [14]	Boolean Join	$(ev+y)n+xt{\cdot}m{\cdot}T$	$O(x\cdot n^{'}+x\cdot\ell_{max})$	$O(logn)+O(\ell_1\cdot\ell_2)$	$O(x\cdot n^{'})+O(\ell_1+\ell_2)$	×	×
FBJXT [16]	Dynamic Join	$(ev+y)n+xt{\cdot}m{\cdot}T$	$O(\ell_1+\ell_2))$	$O(\ell_1+\ell_2)+O(\ell_1\ell_2)$	$O(\ell_1+\ell_2)$	×	×
JXTMM	Join	$1.23n(ev+y+h)+\beta$	$O(\ell_{\max_1}+\ell_{\max_2})$	$O(\ell_{\max_1}+\ell_{\max_2})+O(\ell_{\max_1}\cdot\ell_{\max_2})$	$O(\ell_{\max_1}+\ell_{\max_2})+O(\ell_{\max_1}\cdot\ell_{\max_2})+O(\text{BF})$	✓	✓

Let n denote the number of multi-maps consisting of attribute-value pairs and corresponding identifiers. Let m and T represent the number of records and join attributes in the data table Tab, respectively. ev is the encrypted identifier, y is a blinded value, xt is the cross-tag in XSet and h is the hash value of xt. x denotes the number of conjunctive keywords. ℓ_i is the number of identifiers matching the attribute-value pair w_i (i.e., the volume). $\ell_{\max_1}$ and $\ell_{\max_2}$ represent the maximum volumes across all attribute-value pairs in each table. $|DB(q)|$ denotes the result size of query q. Additionally, β is a small positive integer representing the XOR filter parameter, $n^{'}$ is the size of bucket in TNT-QJ and BF denotes the query complexity in the bloom filter. Details are discussed in Sec. VII.

However, schemes for conjunctive queries on equi-joins of encrypted tables, such as JXT [13], still suffer from several leakage risks. These include the leakage of volume and result patterns in table join query protocols. The result pattern leakage can reveal the value of returned identifiers, while the volume pattern leakage may expose the number of returned identifiers [18]. Several works [19]–[22] have been proposed to reconstruct the plaintext data and infer the queried keywords by analyzing the result pattern or volume pattern. To address this security problem, Kamara et al. [23] developed encrypted multi-maps (EMMs) to hide the volume of values associated with a single key. Their proposed construction can reduce the storage cost at server to $O(n)$ while maintaining $O(\ell \log n)$ query complexity. Since then, a series of EMM-based schemes [17], [18], [24] for volume-hiding query have been developed. However, these approaches can only eliminate volume pattern leakage risks for single keyword search. Similarly, some recent [25], [26] works have attempted to support volume and other pattern hiding in more advanced query settings. [25] achieves a response and volume hiding for conjunctive keyword search using homomorphic encryption and PSI protocols. Another work [26] designs an efficient multi-range query scheme that hides both access and volume patterns via ORE and SGX-based techniques.

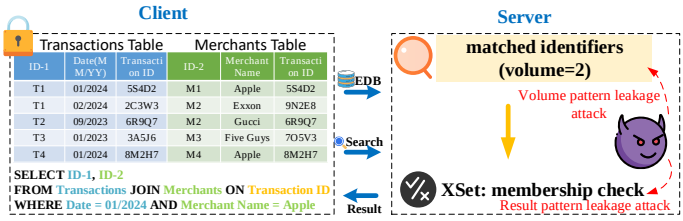


Fig. 1: Leakage risks.

As Fig. 1 illustrated, the leakage risks in JXT is given as follows: (1) The client encrypts the transactions and merchants tables, which are then sent to the server. (2) A query is issued by the client to join the two tables on the "Transaction ID" attribute, and is filtered by specific conditions (e.g., "Date = 01/2024" and "Merchant Name = Apple"). (3) Upon executing the search algorithm, the server can observe the number of identifiers (i.e., the volume) associated with the attribute-value pairs. (4) Once an attacker accumulates a sufficient number of volume observations from the server, it may be able to reconstruct the entire dataset by the data reconstruction attack [27]–[29]. (5) Additionally, the server always performs membership detection by examining the XSet, which may incur result pattern leakage.

Consequently, the problem can be framed as follows: is it possible to design a table join query scheme that eliminating both volume leakage and result leakage attacks?

In this paper, we construct a pattern-hiding table join query scheme, termed JXTMM (JXT multi-map). We compare JXTMM with existing methods in terms of search type, storage cost, client computation cost, server computation cost, and security strength (see TABLE I). Compared with XORMM [17], which supports only single keyword queries with volume-hiding but cannot handle join operations, and with schemes such as JXT [13], TNT-QJ [14], OTJXT [15], FBJXT [16], which support join queries but leaks both volume and result patterns, JXTMM achieves efficient join queries while providing strong pattern-hiding guarantees. The challenges of constructing JXTMM go beyond simply combining JXT and the XOR filter. We leverage the strengths of the XOR filter [30] to design new forms of TSet and XSet. For TSet, we build an EMM by encrypting attribute-value pairs along with their identifiers. Each multi-map is inserted into the XOR filter using a constrained pseudorandom function, and the remaining empty positions are filled with dummy values. For XSet, we map the hash values of "cross-tag" xtags into an XOR filter, and shift the membership checking task to the client. This prevents the server from inferring correlations between xtags during query execution and eliminates result pattern leakage. In this way, JXTMM achieves the first table join query scheme with both volume and result pattern hiding. The main contributions of this paper are summarized as follows:

- To the best of our knowledge, JXTMM is the first pattern-hiding join query scheme. It is designed to eliminate the volume and the result pattern leakage in JXT. JXTMM can also achieve practical efficiency in the query process.

- To suppress the volume pattern leakage, we reconstruct TSet structures for all attribute-value pairs. It can record identifiers that contain the attribute-value pairs in a volume-hiding manner. Specially, we combine the XOR filter and a constrained pseudorandom function with the multi-map (including the attribute-value pairs and identifiers) techniques.
- To eliminate the result pattern leakage, we move the XSet membership checkability task from the server to the client. In details, the server only generates candidate $xtag$ values, while the client is in charge of the $xtag$ membership testing.
- A comprehensive and formal security analysis is conducted to demonstrate the security of JXTMM, including the analysis of its leakage functions, specific volume-hiding properties and adaptive security. Furthermore, we perform JXTMM on a database consisting of two tables. The experimental results have shown that, compared to the JXT [13], JXTMM can achieve volume-hiding table join queries with practical efficiency.

The structure of the paper is outlined as follows. Section II provides a review of related work. Section III introduces the preliminaries. Sections IV presents the formalization of the system model, scheme definition, and security model. Then, our construction is proposed in Section V, followed by the security analysis and the efficient analysis in Section VI and Section VII, respectively. Section VIII presents the performance evaluation and comparison. Section IX discusses the limitations and future work. Finally, Section X concludes the paper.

II. Related Work

A. Symmetric Searchable Encryption

STE has emerged as a fundamental technique in enabling secure and efficient operations on encrypted data. Chase et al. [1] first introduced STE to enable secure query on the encrypted arbitrary data structures [11], [31], such as graphs or relational databases. It has garnered much attention for the widely applications in cloud services, legal surveillance or network provenance [32]–[34]. As a special case of STE, symmetric searchable encryption [35] has been extensively explored from three key perspectives as follows: Firstly, several works focused on *the security improvement*. For instance, Zuo et al. [36] designed a bitmap index, and all files are encoded with binaries to achieving a strong security. Amjad et al. [37] introduced the concept of associative security in SSE, and formally demonstrated the way to defend against various injection attacks based on the associative security. Secondly, *saving the cost of storage, communication or computation in SSE* is another important problem. e.g., Wang et al. [3] proposed a scheme to guarantee the system security by hiding the search pattern without relying on Oblivious RAM. Liu et al. [38] leveraged an order-weighted inverted index and bitmap structure to achieve high search efficiency. A partitioning strategy is also introduced to make it fit to large-scale datasets. At last, researchers also conducted on *extending expressive queries to improve the application flexibility*, such as Chen et al. [39] introduced a semantic-extended ciphertext retrieval method based on the word2vec model, which enables efficient context-aware and semantically range SSE. Li et al. [40] proposed a method for querying with multiple keywords on specific boolean expressions.

B. Secure Table Join Queries

Table join query is important for performing expressive queries on encrypted relational databases, which is hard to be supported by the SSE. Specifically, Kamara et al. [11] proposed a scheme termed SPX based on structured encryption. This approach pre-computes and stores all possible joins in the encrypted database with a heuristic normal form representation. However, SPX incurs significant storage overhead due to the pre-computation of equi-joins. To address this issue, Cash et al. [12] introduced the concept of partially pre-computed joins by offloading certain join operations to the client. Different from prior join query methods, JXT [13] avoids join pre-computation by letting the server perform subqueries on two tables. It verifies the existence of all combinations that match record identifiers and the attribute-value pairs across tables. Wu et al. [14] proposed TNT-QJ, a practical TwiN cross-Tag protocol, which supports arbitrary Boolean queries in both disjunctive and conjunctive normal forms over multi-table joins. This approach reduces storage overhead and improves query efficiency. Xu et al. [15] introduced OTJXT, an equi-join scheme that eliminates the conditional combination pattern leakage of JXT across multiple queries. OTJXT also addresses linear search complexity, yielding significant gains in both search and storage efficiency. Recently, Li et al. [16] introduced a dynamic and secure join query protocol FBJXT for multi-user environments. FBJXT achieves forward and backward security through dynamic oblivious cross-tags, while also resisting key leakage and mitigating collusion attacks by using distributed key-homomorphic PRFs.

C. Leakage Pattern in SSE

The leakage concerns remain challenge for secure queries. Several prior searchable encryption schemes [41]–[43] have revealed certain information about the queries and their corresponding responses. These concerns are generally categorized into three types: search-pattern leakage, result-pattern leakage, and volume-pattern leakage. As [13] mentioned, most prior SSE schemes are vulnerable to the volume-pattern leakage attacks. For instance, Kellaris et al. [20] introduced the first data reconstruction attack based solely on the volume of the search results. However, this attack needs the adversary to perform $O(n^4 \log n)$ queries, and requires the knowledge of underlying query distribution. Then, Grubbs et al. [21] introduced several attack scenarios, which assume either a uniform query distribution or one known distribution to the adversary. Additionally, an AOR (Approximate Order Reconstruction) attack is proposed to instead of value reconstruction. It can achieve the attack goals without relying on strong assumptions. Gui et al. [29] proposed volume attacks which can effectively perform on missing or spurious queries, as well as on the noisy results.

A straightforward approach to mitigate volume-pattern leakage is initial padding. It can make each result size to the same length, which is the maximum volume. However, the padding method always incurs significant storage overhead. Thus, Kamara et al. [18], [23] proposed the first volume-hiding encryption scheme for encrypted keyword searches. It can achieve volume-hiding by padding all query results to a fixed size. In order to further reduce the storage and search overhead, Patel et al. [24] introduced a delegatable PRF volume-hiding encrypted multi-map (dprfMM), which utilizes the cuckoo hashing and delegatable PRFs techniques. But its query communication efficiency still remains suboptimal. Therefore, Wang et al. [17] developed a lossless volume-hiding EMM scheme (XORMM) based on XOR filters to reduce query communication and storage overhead. It also introduced a practical verifiable EMM scheme through an authentication method. Nevertheless, the above schemes are all limited to supporting only single-keyword queries, which significantly restricts their practicality in real-world applications.

III. Preliminaries

In this section, we introduce the fundamental cryptographic primitives that underpin our scheme, namely constrained pseudorandom functions (cPRFs) and encrypted multi-maps (EMMs). To ensure clarity and readability, all mathematical notations used throughout the paper are summarized in Table II, which specifies both their meaning and domain information. This unified notation table serves as a reference point for the subsequent sections, avoiding ambiguity and enhancing logical consistency.

TABLE II: Notations.

Notations	Descriptions
$\mathcal{K}$	the PRF master key space
$\mathcal{X}$, $\mathcal{Y}$	the PRF input domain and the PRF output range
λ	a security parameter
$k \in \mathcal{K}$	a PRF master key
f	a random function
$\mathcal{P}$	the family of predicates
$\mathcal{K}_c$	the constrained key space
$k_p \in \mathcal{K}_c$	a constrained key associated with predicate p
MM	the multi-map storing key/value pairs
$\mathbf{K}$	the multi-map key space
$\mathbf{V}$	the multi-map value space
$\overrightarrow{v[c]}$	the c-th element of the value vector
q	a query
K_d	a cPRF key
K_e, K_I, K_W, K_Z, K'_Z and K_H	symmetric keys with the length λ
w	the attribute-value pair
$\overrightarrow{ind}$	the identifier
i	the order of a multi-map $(w_i, \overrightarrow{ind_i})$ in MM
j	the order of $\overrightarrow{ind_i}$ mapped by w_i
DB	the relational database
$attr^*$	the common attribute value in the DB
EMM, XMM	the XOR filters
EDB	the encrypted database
h_0, h_1	two different hash functions
BF	the bloom filter

A. Constrained Pseudorandom Function

We begin with the notion of pseudorandom functions (PRFs) and then extend it to constrained pseudorandom functions (cPRFs), which reduce computation complexity in our scheme.

A pseudorandom function (PRF) F: $\mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$. Its advantage for any adversary $\mathcal{A}$ in probabilistic polynomial time (PPT), is as follows:

$$Adv^{PRF}_{\mathcal{A},F}(\lambda) = |\Pr[\mathcal{A}^{F(k,\cdot)}(\lambda) = 1] - \Pr[\mathcal{A}^{f(\cdot)}(\lambda) = 1]| \quad (1)$$

It is negligible in λ, where $k \xleftarrow{\$} K$ and f is a random function from $\mathcal{X}$ to $\mathcal{Y}$. Specifically, when $\mathcal{X} = \mathcal{Y}$, we call F a pseudorandom permutation.

To enable fine-grained control, we consider a family of predicates $\mathcal{P} = \{p : \mathcal{X} \rightarrow \{0,1\}\}$, a PRF F: $\mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ is constrained if there is an additional key space $\mathcal{K}_c$ and two additional algorithms **F.Cons** and **F.Eval**:

(1) **F.Cons**(k, p): it takes a PRF key $k \in \mathcal{K}$, the description of a predicate $p \in \mathcal{P}$ as input, and outputs a constrained key $k_p \in \mathcal{K}_c$. The key k_p enables the evaluation of $F(k, x)$ for all $x \in \mathcal{X}$ s.t. $p(x) = 1$.

(2) **F.Eval**(k_p, x): it takes a constrained key $k_p \in \mathcal{K}_c$ and an $x \in \mathcal{X}$ as input. If k_p is the output of **F.Cons**(k, p) for a PRF key $k \in \mathcal{K}$ then **F.Eval**(k_p, x) outputs

$$\textbf{F.Eval}(k_p, x) = \begin{cases} F(k, x), & \text{if} \quad p(x) = 1 \\ \perp, & \text{otherwise} \end{cases} \quad (2)$$

where $\perp \notin \mathcal{Y}$.

Security intuition. Informally, a cPRF ensures that the function values revealed under constrained keys remain indistinguishable from random values at all other points [44]. Formally:

Definition 1. *A PRF F is constrained secure with respect to $\mathcal{P}$ if for all PPT adversaries $\mathcal{A}$, its advantage*

$$Adv^{PRF}_{\mathcal{A},F}(\lambda) = \left| \Pr[\text{Exp}^{PRF}(1) = 1] - \Pr[\text{Exp}^{PRF}(0) = 1] \right|$$

is negligible.

The experiment $\text{Exp}^{PRF}(b)$ is structured as follows. A random key $k \in \mathcal{K}$ is sampled, and three empty sets $P, V, C \subseteq \mathcal{X}$ are initialized: V records evaluated points, C records challenge points, and P records predicates. The invariant $C \cap V = \emptyset$ prevents trivial attacks. The adversary $\mathcal{A}$ can issue the following queries:

- **Evaluation**(x): If $x \notin C$, return $F(k, x)$ and update $V \leftarrow V \cup \{x\}$.
- **Constrained-key**(p): If $p(x) = 0$ for all $x \in C$, return $k_p =$ **F.Cons**(k, p) and update $P \leftarrow P \cup \{p\}$.
- **Challenge**(x^*): If $x^* \notin V$ and $p(x^*) = 0$ for all $p \in P$, then return $F(k, x^*)$ if $b = 0$, or a random $y \in \mathcal{Y}$ if $b = 1$. Update $C \leftarrow C \cup \{x^*\}$.

Finally, $\mathcal{A}$ outputs a bit b', which is also the experiment's output.

B. Encrypted Multi-Maps

Next, we recall the encrypted multi-map (EMM), a key data structure enabling efficient query processing on encrypted databases.

A *multi-map* stores key/value pairs $\text{MM} = \{(key, \overrightarrow{v})\}$, where $key \in \mathbf{K}$ and $\overrightarrow{v[c]} \in \mathbf{V}$ denotes the c-th entry of the value vector associated with key [18], [23]. It supports:

- **MM.Get**(key): return $\overrightarrow{v} = \text{MM}[key]$.

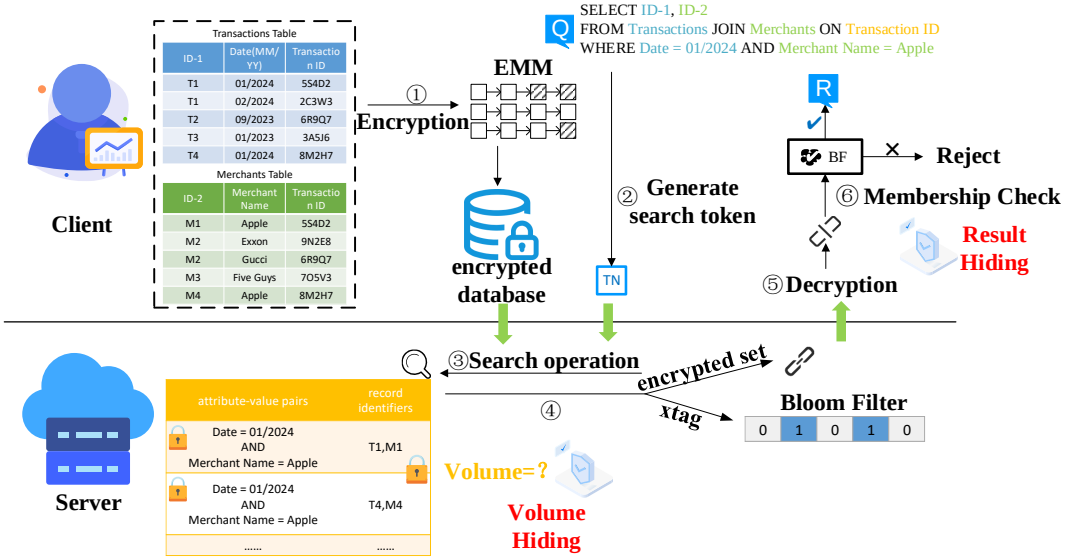


Fig. 2: System model overview.

- **MM.Put**$(key, \overrightarrow{v})$: store $(key, \overrightarrow{v})$ in MM.

To secure queries, we use an *encrypted multi-map* (EMM), defined as $\text{EMM} = (\textbf{Setup}, \textbf{Search})$:

- **Setup**$(1^\lambda, \text{MM})$: On input a multi-map MM and security parameter λ, output (EMM, K) where K is a secret key.
- **Search**$(K, q; \text{EMM})$: Using K and query q, produce a token tk_q for the server. The server searches EMM with tk_q and returns the encrypted result $\text{EMM}[q]$, which the client decrypts using K to obtain $\text{MM}[q]$.

IV. Problem Formulation

In this section, we introduce the formal problem setting of our scheme. We begin with the system model and query workflow, then describe the corresponding security model, and finally specify the leakage patterns that will be considered in our analysis. To avoid ambiguity, all mathematical symbols used below are explicitly defined in the table of Notations (see Section III).

A. System Model

Our system consists of two entities: a client and a server, as illustrated in Fig. 2.

- The client owns a relational database

$$DB = \{\text{MM}_{Tab}\}_{Tab \in [N]},$$

  where each MM_{Tab} is a multi-map table containing attribute-value pairs and their associated record identifiers.
- Formally, each table is represented as

$$\text{MM}_{Tab} = \{(w, \overrightarrow{ind})\},$$

  where w denotes an attribute-value pair and $\overrightarrow{ind}$ the corresponding set of record identifiers.

Due to limited local resources, the client outsources the encrypted database to a remote server. The query processing workflow proceeds as follows:

- **Step** ①: The client encrypts each table, mapping attribute-value pairs and corresponding record identifiers into an encrypted multi-map (EMM), and uploads the encrypted database to the server.
- **Step** ②: For a query set Q, the client generates search tokens (TN) and sends them to the server.
- **Step** ③: Upon receiving a token, the server retrieves candidate attribute-value pairs and their record identifiers. The scheme ensures that the volume of identifiers remains hidden, as all attribute-value pairs are padded to the same size.
- **Step** ④: The server stores the $xtag$s corresponding to the retrieved identifiers in a Bloom filter to verify valid matches. It then returns the Bloom filter along with the encrypted result set to the client.
- **Steps** ⑤ **&** ⑥: The client decrypts the result set and checks membership using the Bloom filter, accepting only verified results.

This design allows table join queries over encrypted data without pre-computation, while hiding volume and result patterns to mitigate potential leakage risks.

B. Security Model

To capture the adversarial capabilities, we define the security of our scheme through two games: a *real-world* game and an *ideal-world* game.

Let $\sum = (\text{Setup}, \text{Search})$ denote the JXTMM scheme, $\mathcal{A}$ a probabilistic polynomial-time (PPT) adversary, and $\mathcal{S}$ a simulator. If $\mathcal{A}$ cannot distinguish between the real and ideal games with non-negligible probability, then the scheme is secure with respect to the leakage function $\mathcal{L}$.

- **Real**$_{\mathcal{A}}^{\sum}(\lambda)$: The adversary $\mathcal{A}$ selects a table MM_{Tab}. The **Setup** algorithm generates the encrypted database EXMM, which is given to $\mathcal{A}$. Then $\mathcal{A}$ adaptively issues queries. For each query q, the **Search** algorithm is executed and the transcript is returned. Finally, $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$, indicating its guess.

- **Ideal**$^{\sum}_{\mathcal{A},\mathcal{S}}(\lambda)$: The adversary $\mathcal{A}(1^\lambda)$ selects a table MM_{Tab}. The simulator $\mathcal{S}$ generates EXMM based only on the leakage function $\mathcal{L}(\mathrm{MM}_{Tab})$, which is returned to $\mathcal{A}$. For each query q, $\mathcal{S}(\mathcal{L}(\mathrm{MM}_{Tab}, q))$ produces the transcript. Finally, $\mathcal{A}$ outputs a bit b as in the real game.

Definition 2. *A JXTMM scheme* $\sum = (Setup, Search)$ *is* $\mathcal{L}$*-adaptively secure if, for every stateful honest-but-curious PPT adversary* $\mathcal{A}$*, there exists an efficient simulator* $\mathcal{S}$ *such that*

$$\left|\Pr[Real^{\sum}_{\mathcal{A}}(\lambda)=1]-\Pr[Ideal^{\sum}_{\mathcal{A},\mathcal{S}}(\lambda)=1]\right| < negl(\lambda).$$

Here, $\mathcal{A}$ may make a polynomial number of queries $poly(\lambda)$ against the challenger $\mathcal{C}$.

C. Leakage Patterns

Finally, we describe the potential leakage captured in our model, following the framework in [13]. Suppose that h queries are executed, denoted by the history $q=(q_1,\ldots,q_h)$, where each q_i is a join query. The leakage function $\mathcal{L}$ reveals the following patterns:

- **Domain Size** ($dsize$): Total number of attribute-value pairs and identifiers in a table.
- **Result Pattern** (rp): The set of identifiers matching each query.
- **Response Length** ($rlen$): Number of identifiers associated with each attribute-value.
- **Query Equality** (qeq): Indicates whether two queries are identical (search pattern).
- **Maximum Response Length** ($mrlen$): Maximum number of identifiers associated with any attribute-value.
- **Join Attribute Distribution** (jd): Frequency distribution of join attribute values $attr^*$ in the first table.
- **Conditional Intersection** (ip): Intersection of identifiers between two distinct join queries.

V. Construction of JXTMM

In this section, we propose the volume-hiding and result-hiding table join query scheme (JXTMM). A technical overview of the JXTMM scheme is provided at first, and then the specific details of the scheme are given.

A. Technical Overview

The main idea of JXTMM scheme is to store the table MM_{Tab} in a multi-map form, which contains attribute-value pairs w_i and their corresponding identifiers $\overrightarrow{ind_i}$.

Specifically, the encryption algorithm inserts the multi-map $\mathrm{MM}_{Tab}=\{(w_i,\overrightarrow{ind_i})\}$ into the XOR filter. Each w_i can be associated to one or more ind_i, and the order of the ind_i is denoted as indices j, where $j\in[\ell_{\max}]$. Then, by performing the XOR filter's mapping steps, each attribute-value/identifier pair $(w_i, ind_i[j])$ together with an index k of a temporary array (exactly, $((w_i,j),k)$) is pushed into a stack sequentially. After that, the encryption of the value $ind_i[j]$ associated with each pair $((w_i,j),k)$ in the stack is inserted into the EMM.

In particular, the ciphertext $Enc(K_e, ind_i[j])$ is placed into the EMM by setting $\mathrm{EMM}[k]=\mathrm{EMM}[F_{K_d}(w_i||j||1)]\oplus\mathrm{EMM}[F_{K_d}(w_i||j||2)]\oplus Enc(K_e,ind_i[j])$, where $k=F_{K_d}(w_i||j||0)$, K_e is a symmetric encryption key, and K_d is cPRF key to reduce the communication complexity. Based on [45], when the XOR filter is mapped 3 times, the storage space of the EMM is minimized to $\lfloor 1.23n\rfloor+\beta$. Then, a concrete example of the XOR filter is given in Fig. 3. In the figure, the "Temporary Array T" is a temporary array for $((w_i,j),k)$, while "Array B" represents the EMM. The client can query any attribute-value pair w_i, by simply sending the 3ℓ PRF values $\{F_{K_d}(w_i||j||0),\ F_{K_d}(w_i||j||1),\ F_{K_d}(w_i||j||2)\}_{j\in\ell_{\max}}$ as a search token to the server. Upon receiving the token, the server can directly retrieve corresponding values $\{\mathrm{EMM}[F_{K_d}(w_i||j||t)]\}_{t\in\{0,1,2\}}$ for each $j\in\ell_{\max}$, and at last obtains the final $\ell_{\max}$ results by performing the XOR operation. In this way, the server will return search results for all queries with the same volume $\ell_{\max}$, thereby achieving volume-hiding.

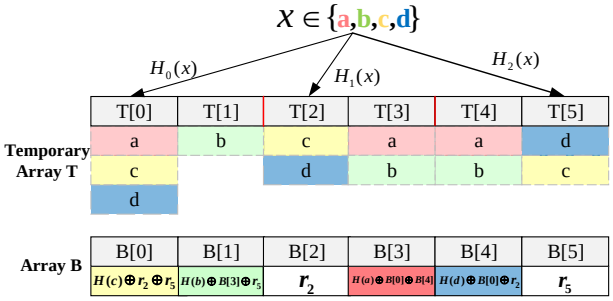


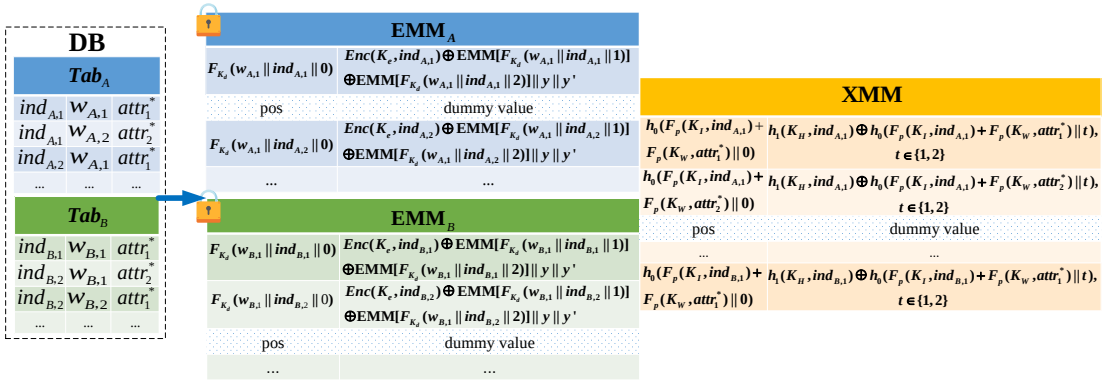
Fig. 3: XOR filter.

To prevent result pattern leakage and obfuscate the exact number of identifiers, our JXTMM scheme modifies the structure of XSet. Specifically, the JXTMM scheme constructs a structure termed XMM with an XOR filter. It employs a symmetric hash key K_H, a hash function h_0 with an output range of $\lfloor 1.23n\rfloor+\beta$ and a hash function h_1, and stores the hash value $h_1(K_H, w||ind_i[j])\oplus h_0(F_p(K_I, ind_i[j])+F_p(K_W, attr^*)||1)\oplus h_0(F_p(K_I, ind_i[j])+F_p(K_W, attr^*)||2)$ at the position $h_0(F_p(K_I, ind_i[j])+F_p(K_W, attr^*)||0)$. Consequently, the server is unable to distinguish the values between genuine hashes and spurious entries. Then, all hash values are sent to the client. Additionally, we integrate these hashes into the bloom filter to enhance the efficiency of the response overhead.

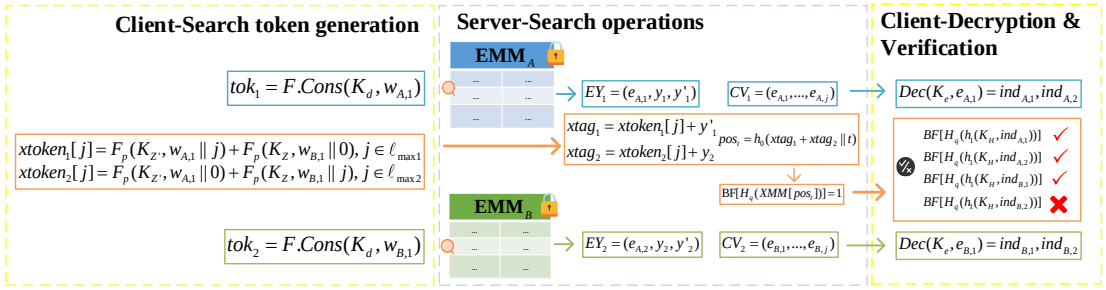
B. Building Blocks

In this section, we present the construction of the JXTMM scheme. It mainly consists of two polynomial-time algorithms: **JXTMM.Setup** (Algorithm 1) and **JXTMM.Search** (Algorithm 2). Fig. 4 provides an overview of the encrypted multi-map construction and join query process. The following content present a detailed description of each step, including token generation, server-side retrieval with bloom filter checks, and client-side decryption and verification. A detailed description is provided below.

Setup$(1^\lambda, \mathbf{MM_{tab}}, attr^*)\rightarrow(K, \mathbf{EXMM})$: As shown in Algorithm 1, the Setup algorithm is executed by the client, with inputs including a security parameter λ, a multi-map $\mathrm{MM}_{\mathrm{tab}}$ for each table, and a common join attribute $attr^*$.



(a) Example of the setup algorithm.



(b) Example of the search algorithm.

Fig. 4: Illustration of algorithm operations. Take the query statement "SELECT * FROM Tab_A JOIN Tab_B ON $attr^* = attr_1^*$ WHERE $w_A = w_{A,1}$ AND $w_B = w_{B,1}$" as an example.

Each table is processed independently, so only one MM_{tab} is handled at a time. The output is a set of symmetric keys K and data structures $\text{EXMM} = (\text{EMM}, \text{XMM})$. An example of encrypted database construction is shown in Fig. 4a. The original multi-maps Tab_A and Tab_B are first processed to form encrypted multi-maps EMM_A and EMM_B. Each attribute-value-identifier pair (e.g., (w_{A_1}, ind_{A_1})) is mapped using the cPRF F_{K_d}, combined with auxiliary values (y, y'), and encrypted as $Enc(K_e, ind_{A_1})$. The resulting entries are stored in EMM, while empty slots are filled with dummy values to ensure consistency. Next, XMM is constructed: each identifier ind_{A_1} together with the join attribute $attr^*$ is processed via F_p functions, and the results are mapped into positions using the hash function h_0. To further obfuscate the identifiers, h_1 is applied, and dummy values are inserted to conceal the true distribution. The details of the **Setup** algorithm are as follows:

- **Scheme parameters**: As demonstrated in Algorithm 1, the client randomly selects a constrained PRF (cPRF) key K_d and a set of symmetric keys with the length λ, i.e., K_e, K_I, K_W, K_Z, K'_Z, and K_H. Then it initializes the following structures into empty sets, including $xlist$, two XOR filters EMM_{Tab} and XMM, two stacks S_1 and S_2, two queues Q_1 and Q_2, two temporary arrays T_1 and T_2.
- **Encryption parameters**: For the encryption procedure, the client first generates a set of "padding elements" in the form of $z_0 = F_p(K_z, w_i||0)$, $z'_0 = F_p(K_{z'}, w_i||0)$ on each attribute-value pair w_i. Then it iterates over the identifiers corresponding to each attribute-value pair. The additional "padding elements" and "cross-tag" are calculated as:

$$\begin{aligned} z_{cnt} &= F_p(K_z, w_i||j) \\ z'_{cnt} &= F_p(K_{z'}, w_i||j) \\ xlist[i][j] &= F_p(K_I, ind_i[j]) + F_p(K_W, attr^*) \end{aligned} \tag{3}$$

Based on the "padding elements", the client computes the "blinded values" y and y' to protect identifiers and attribute-value pairs:

$$\begin{aligned} y &= F_p(K_I, ind_i[j]) - (z_0 + z_{cnt}) \\ y' &= F_p(K_W, attr^*) - (z'_0 + z'_{cnt}) \end{aligned} \tag{4}$$

- **EMM encryption**: (1) The algorithm utilizes a temporary array T to store the attribute-value-identifier pairs $(w_i, ind_i[j])$ from the multi-map MM. Each pair is inserted into the T at an index k determined by the cPRF $F_{K_d}(w_i||j||0)$. (2) Once all pairs are inserted, the algorithm searches T for entries $T[k]$. When such an entry is found, the pair and its index k are removed from T and pushed into the stack S. This process repeats until all pairs in MM are processed. (3) The stack S then holds successfully processed entries from T, denoted by mapped-temporary index pairs $((w_i, j), k)$. Subsequently, the encrypted values $ind_i[j]$ of each pair, along with the y and y' are inserted into the EMM. For each pair, the algorithm sets:

$$\begin{aligned} \text{EMM}[k] = &\text{EMM}[F_{K_d}(w_i||j||0)] \oplus \text{EMM}[F_{K_d}(w_i||j||1)] \oplus \\ &\text{EMM}[F_{K_d}(w_i||j||2)] \oplus Enc(K_e, ind_i[j]) \end{aligned} \tag{5}$$

where $Enc(K_e, ind_i[j])$ is a ciphertext. Finally, empty slots in the EMM are filled with dummy values to ensure the consistency and integrity of the multi-map.

Algorithm 1 The Setup of JXTMM

Setup$(1^\lambda, \mathrm{MM}_{Tab}, attr^*)$:

Input: $1^\lambda, \mathrm{MM}_{Tab}, attr^*$
Output: $(K,$ EXMM$)$

$K_d, K_e, K_I, K_W, K_Z, K_{Z'}, K_H \leftarrow \{0,1\}^\lambda$
$\mathrm{EMM}_{Tab}, \mathrm{XMM} \leftarrow \emptyset$
$S_1, S_2, Q_1, Q_2, T_1, T_2 \leftarrow \emptyset$
$xlist \leftarrow \emptyset$

// Encrypt parameters
for $i \in [n]$ **do**:
  $z_0 = F_p(K_z, w_i||0)$, $z'_0 = F_p(K_{z'}, w_i||0)$
  for $j \in [|\overrightarrow{ind_i}|]$ **do**
    $z_{cnt} = F_p(K_z, w_i||j)$, $z'_{cnt} = F_p(K_{z'}, w_i||j)$
    $xind = F_p(K_I, ind_i[j])$, $xw = F_p(K_W, attr^*)$
    $xlist[i][j] = xind + xw$
    $y = xind - (z_0 + z_{cnt})$, $y' = xw - (z'_0 + z'_{cnt})$

// Insert MM_{Tab} into XOR filters
for $i \in [n]$ and $j \in [|\overrightarrow{ind_i}|]$ **do**
  for t = 0 to 2 **do**
    $T_1[F_{K_d}(w_i||j||t)] \leftarrow (w_i, ind_i[j])$
    $pos \leftarrow h_0(xlist[i][j]||t)$
    $T_2[pos] \leftarrow (w_i, ind_i[j])$
for k = 0 to $|T_1|$ **do**
  if $|T_1[k]| = 1$ **then**
    $Q_1 \leftarrow k$
  if $|T_2[k]| = 1$ **then**
    $Q_2 \leftarrow k$
while $Q_1 \neq null$ **do**
  $k \leftarrow Q_1$
  $(w_i, ind_i[j]) \leftarrow T_1[k]$
  $S_1 \leftarrow ((w_i, ind_i[j]), k)$
  for t = 0 to 2 **do**
    $T_1[F_{K_d}(w_i||j||t)] \leftarrow T_1[F_{K_d}(w_i||j||t)]/(w_i, ind_i[j])$
    if $T_1[F_{K_d}(w_i||j||t)]| = 1$ **then**
      $Q_1 \leftarrow F_{K_d}(w_i||j||t)$
while $Q_2 \neq null$ **do**
  $k \leftarrow Q_2$
  $(w_i, ind_i[j]) \leftarrow T_2[k]$
  $S_2 \leftarrow ((w_i, ind_i[j]), k)$
  for t = 0 to 2 **do**
    $T_2[pos] \leftarrow T_2[pos]/(w_i, ind_i[j])$
    if $T_2[pos]| = 1$ **then**
      $Q_2 \leftarrow pos$
if $(|S_1| \neq |\mathrm{MM_{Tab}}|)||(|S_2| \neq |\mathrm{MM_{Tab}}|)$ **then**
  return Failure

// Encrypt the associated value in XOR filters
for $((w_i, ind_i[j], k)) \in S_1$ **do**
  $\mathrm{EMM}_{Tab}[k] \leftarrow Enc(K_e, ind_i[j]), y, y'$
  for t = 0 to 2 **do**
    if $F_{K_d}(w_i||j||t) \neq k$ **then**
      if $\mathrm{EMM}_{Tab}[F_{K_d}(w_i||j||t)] = null$ **then**
        $\mathrm{EMM}_{Tab}[F_{K_d}(w_i||j||t)] \leftarrow \{0,1\}^{|\gamma|}$
      $\mathrm{EMM}_{Tab}[k] \leftarrow \mathrm{EMM}_{Tab}[k] \oplus \mathrm{EMM}_{Tab}[F_{K_d}(w_i||j||t)]$
for $((w_i, ind_i[j], k)) \in S_2$ **do**
  $\mathrm{XMM}[k] \leftarrow h_1(K_H, ind_i[j])$
  for t = 0 to 2 **do**
    if $pos \neq k$ **then**
      if $\mathrm{XMM}[pos] = null$ **then**
        $\mathrm{XMM}[pos] \leftarrow \{0,1\}^{|\gamma|}$
      $\mathrm{XMM}[k] \leftarrow \mathrm{XMM}_{Tab}[k] \oplus \mathrm{XMM}[pos]$

// Pad with dummies
for $j \in [|\mathrm{EMM}_{Tab}|]$ **do**
  if $\mathrm{EMM}_{Tab}[j] = null$ **then**
    $\mathrm{EMM}_{Tab}[j] \leftarrow \{0,1\}^{|\gamma|}$
  if $\mathrm{XMM}[j] = null$ **then**
    $\mathrm{XMM}[j] \leftarrow \{0,1\}^{|\gamma|}$
$K \leftarrow (K_d, K_e, K_I, K_W, K_Z, K_{Z'}, K_H)$
EXMM $\leftarrow$ (EMM, XMM)
return $(K,$ EXMM$)$

- **XMM encryption**: The encryption process for XMM is similar to that of EMM. However, to prevent the server from distinguishing real hash entries from fake ones, the mapping positions in XMM are obfuscated by the hash function $h_0(xlist[i][j]||t)$. Furthermore, a different hash function h_1 is performed on the identifier $ind_i[j]$ to further hide the underlying data. Finally, the client sends the key set K along with the encrypted table EXMM $\leftarrow$ (EMM, XMM) to the server.

Search$(K,\ q(w_1,\ w_2,\ Tab_1,\ Tab_2,\ attr^*),$ EXMM$) \rightarrow (\mathrm{MR}_w)$: As depicted in Algorithm 2, the search algorithm is jointly executed by the client and server. The client inputs a table join query q and the key set K, while the server, inputs the set of EXMMs. The outputs are two sets of identifiers MR_w. An illustrative example is shown in Fig. 4b. Given query keywords $w_{A,1}$ and $w_{B,1}$, the client derives keys $(K_d, K_Z, K_{Z'})$ to generate search tokens (tok_1, tok_2) and arrays of join tokens $(xtoken_1, xtoken_2)$. The server uses these tokens to retrieve ciphertext sets EY_{w_i} containing blind values, compute candidate $xtag$ values, update positions pos_t in the Bloom filter, and return ciphertext sets CV and BF. Finally, the client decrypts CV with K_e, verifies matches via BF, and outputs the valid result. The **Search** algorithm is described in detail as follows:

- **Step 1. Search token generation:** (1) The client derives keys K_d, K_Z and K'_Z from the set of symmetric keys K to generate tokens. (2) Search tokens (tok_{w_1} and tok_{w_2}) are generated for the input attribute-values w_1 and w_2. The cPRF function **F.Cons** is adopted for the generation, which takes K_d as the input key. (3) Next, all indices j in the range $[1, \ell_{\max}]$ are iterated. For each j, the join token arrays $xtoken_{Tab_1}[j]$ and $xtoken_{Tab_2}[j]$ are computed with the the PRF function F_p:
$$\begin{aligned} xtoken_{Tab_1}[j] &= F_p(K_{Z'}, w_1||j) + F_p(K_Z, w_2||0) \\ xtoken_{Tab_2}[j] &= F_p(K_{Z'}, w_1||0) + F_p(K_Z, w_2||j) \end{aligned} \quad (6)$$
(4) Finally, the client aggregates the computed search tokens and join tokens (tok_{w_1}, tok_{w_2}, $xtoken_{Tab_1}$, $xtoken_{Tab_2}$). The results are sent to the server for further processing.
- **Step 2. Search operations:** (1) Upon receiving the search tokens, the server generates $F_{K_d}(w_i||j||t)$ using the **F.Eval** function. (2) Then, the set of ciphertexts EY_{w_i} containing y are updated by the XOR results of the encrypted $F_{K_d}(w_i||j||t)$ and $\mathrm{EY}_{w_i}[j]$. (3) Subsequently, an encrypted value (e_i) is derived from EY_{w_i} to form the ciphertext sets CV. (4) To get pos_t, the $xtag_1$ and

Algorithm 2 The Search of JXTMM

Search: $(K,\ q(w_1, w_2, Tab_1, Tab_2, attr^*)$; EXMM)

Client (Search token generation):

Input: $(K,\ q)$
Output: $(tok_{w_1},\ tok_{w_2},\ xtoken_{t_1},\ xtoken_{t_2})$
$(K_d, K_Z, K'_Z) \leftarrow K$
$tok_{w_1} \leftarrow$ **F.Cons**(K_d, w_1), $tok_{w_2} \leftarrow$ **F.Cons**(K_d, w_2)
$xtoken_{Tab_1}, xtoken_{Tab_2} \leftarrow \emptyset$
for $j \in [\ell_{\max_1}]$ **do**
  $xtoken_{Tab_1}[j] = F_p(K_{Z'}, w_1||j) + F_p(K_Z, w_2||0)$
for $j \in [\ell_{\max_2}]$ **do**
  $xtoken_{Tab_2}[j] = F_p(K_{Z'}, w_1||0) + F_p(K_Z, w_2||j)$
return $(tok_{w_1},\ tok_{w_2},\ xtoken_{t_1},\ xtoken_{t_2})$

Server (Search operations):

Input: $(K, q, tok_{w_1}, tok_{w_2}, xtoken_{t_1}, xtoken_{t_2}$, EXMM)
Output: CV, BF, $\{H_q\}_{q\in[c]})$
$\mathrm{EY}_{w_1}, \mathrm{EY}_{w_2}, \mathrm{CV} \leftarrow \emptyset$
$(K_d) \leftarrow K$
$(\mathrm{EMM}_{Tab_1}, \mathrm{EMM}_{Tab_2}, \mathrm{XMM}) \leftarrow \mathrm{EXMM}$
$\mathrm{BF} \leftarrow 0^b, \{H_q\}_{q\in[c]} : \{0,1\}^\Psi \rightarrow [b]$

// Search sets containing encrypted and blind values
for $j \in [\ell_{\max_1}]$ **do**
  for $t \in [0, 2]$ **do**
    $F_{K_d}(w_1||j||t) \leftarrow$ **F.Eval**$(tok_{w_1}, j||t)$
    $\mathrm{EY}_{w_1} \leftarrow \mathrm{EY}_{w_1}[j] \oplus \mathrm{EMM}[F_{K_d}(w_1||j||t)]$
for $j \in [\ell_{\max_2}]$ **do**
  for $t \in [0, 2]$ **do**
    $F_{K_d}(w_2||j||t) \leftarrow$ **F.Eval**$(tok_{w_2}, j||t)$
    $\mathrm{EY}_{w_2} \leftarrow \mathrm{EY}_{w_2}[j] \oplus \mathrm{EMM}[F_{K_d}(w_2||j||t)]$

// Compute and store cross-tags
for $j_1 \in [\ell_{\max_1}]$ **do**
  $(e_1, y_1, y'_1) \leftarrow \mathrm{EY}_{w_1}[j_1]$
  $xtag_1 = xtoken_{Tab_1}[j_1] + y'_1$
  for $j_2 \in [\ell_{\max_2}]$ **do**
    $(e_2, y_2, y'_2) \leftarrow \mathrm{EY}_{w_2}[j_2]$
    $\mathrm{CV} \leftarrow (e_1, e_2)$
    $xtag_2 = xtoken_{Tab_2}[j_2] + y_2$
    for $t \in [0, 2]$ **do**
      $pos_t = h_0(xtag_1 + xtag_2||t)$
      for $q \in [c]$ **do**
        $\mathrm{BF}[H_q(\mathrm{XMM}[pos_t])] \leftarrow 1$
return $(\mathrm{CV}, \mathrm{BF}, \{H_q\}_{q\in[c]})$

Client (Decryption & verification):

Input: $(K,\ q,\ \mathrm{CV},\ \mathrm{BF},\ \{H_q\}_{q\in[c]})$
Output: $(MR_{w_1},\ MR_{w_2})$
$(K_e, K_d, K_H) \leftarrow K$
$(w_1, w_2, Tab_1, Tab_2, attr^*) \leftarrow q$
$\mathrm{R}_w, \mathrm{MR}_w \leftarrow \emptyset$

// Decrypt the encrypted result sets
for $j_1 \in [\ell_{\max_1}]$ **do**
  if $Dec(K_e, \mathrm{CV}[j_1].e_1) \neq \perp$ **then**
    for $j_2 \in [\ell_{\max_2}]$ **do**
      if $Dec(K_e, \mathrm{CV}[j_2].e_2) \neq \perp$ **then**
        $\mathrm{R}_w = \mathrm{R}_w \cup \{Dec(K_e, \mathrm{CV}[j_1].e_1), Dec(K_e, \mathrm{CV}[j_2].e_2)\}$

// Check the membership
for $(val_1, val_2) \in \mathrm{R}_w$ **do**
  $tmp \leftarrow true$
  for $q \in [c]$ **do**
    $xkv_1 \leftarrow h_1(K_H, val_1)$, $xkv_2 \leftarrow h_1(K_H, val_2)$
    if $\mathrm{BF}[H_q(xkv_1)] \neq 1$ || $\mathrm{BF}[H_q(xkv_2)] \neq 1$ **then**
      $tmp \leftarrow false$
      $Break$
  if $tmp \leftarrow true$ **then**
    $\mathrm{MR}_w \leftarrow (val_1, val_2)$
return MR_w

$xtag_2$ are calculated by the join tokens and the y_i, y'_i from EY_{w_i}:

$$\begin{aligned} xtag_1 &= xtoken_{Tab_1}[j] + y'_1 \\ xtag_2 &= xtoken_{Tab_2}[j] + y_2 \\ pos_t &= h_0(xtag_1 + xtag_2||t) \end{aligned} \tag{7}$$

(5) The bloom filter BF is updated based on the $\mathrm{XMM}[pos_t]$, which are further processed by the hash function $\{H_q\}_{q\in[c]}$. (6) Finally, the server returns the encrypted value sets CV, the bloom filter BF, and the set of hash values $\{H_q\}_{q\in[c]}$ to the client.

- **Step 3. Decryption & verification:** (1) Upon receiving CV from the server, the client decrypts the identifiers to obtain R_w. (2) Each identifier in R_w is verified to determine whether it satisfies the join conditions:

$$\mathrm{BF}[H_q(h_1(K_H, val))] = 1, val \in \mathrm{R}_w \tag{8}$$

Then, the satisfied ones are added to the result sets MR_w. (3) Finally, the record identifiers from MR_w is obtained.

VI. Security Analysis

A. Leakage Analysis

As mentioned in Section IV, the security of our proposed scheme is determined based on the leakage function. Therefore, before analyzing the security, we first define the leakage $\mathcal{L} = (\mathcal{L}_{Setup}, \mathcal{L}_{Search})$ of the JXTMM scheme as follows:

(1) During the setup phase, the scheme reveals $1.23n + \beta$ encrypted attribute-value pairs, along with blind values and hash values, such that $\mathcal{L}_{Setup} = dsize(\mathrm{MM}) = n$. (2) For queries within the encrypted multi-map, it exposes query equality qeq and the maximum volume length $mrlen$ by analyzing the search tokens and results. (3) Additionally, to handle the public attribute $attr^*$ set for table join queries, it leads to the leakage of the join attribute distribution pattern jd. (4) Furthermore, the conditional intersection pattern ip, i.e., $\mathcal{L}_{Search} = (qeq, mrlen, jd, ip)$, is also leaked.

Algorithm 3 Simulator of JXTMM

$\mathcal{S}$.SimSetup
Input: (1^λ, $dsize$)
Output: ($\perp$, EXMM)
$n \leftarrow dsize$
EMM,XMM $\leftarrow \emptyset$, where |EMM|=|XMM|=$\lfloor 1.23n \rfloor+\beta$
for $i = 0$ to |EMM| **do**
  $e \xleftarrow{\$} \{0,1\}^\lambda$
  $(y, y') \xleftarrow{\$} Z_p^*$
  EMM$[i] \leftarrow (e, y)$
  XMM$[i] \xleftarrow{\$} \{0,1\}^\lambda$
EXMM $\leftarrow$ (EMM, XMM)
return ($\perp$, EXMM)

$\mathcal{S}$.SimSearch
Input: (1^λ, qeq, $mrlen$, jd, ip)
Output: ($State$, $State[\psi(q_j)]$)
Using $qeq(\psi(q_1), \psi(q_2), ..., \psi(q_j))$ to analyze $\psi(q_j)$
$w_{j,1}, w_{j,2} \leftarrow \psi(q_j)$
for $i = 1$ to j **do**
  $w_{i,1}, w_{i,2} \leftarrow \psi(q_i)$
  if $w_{i,1} = w_{j,1}$ **then**
    $State[\psi(q_j)][w_{j,1}] \leftarrow State[\psi(q_i)][w_{i,1}]$
  if $w_{i,2} = w_{j,1}$ **then**
    $State[\psi(q_j)][w_{j,1}] \leftarrow State[\psi(q_i)][w_{i,2}]$
  if $w_{i,1} = w_{j,2}$ **then**
    $State[\psi(q_j)][w_{j,2}] \leftarrow State[\psi(q_i)][w_{i,1}]$
  if $w_{i,2} = w_{j,2}$ **then**
    $State[\psi(q_j)][w_{j,2}] \leftarrow State[\psi(q_i)][w_{i,2}]$
if $State[\psi(q_j)][w_{j,1}] = null$ **then**
  $State[\psi(q_j)][w_{j,1}] \xleftarrow{\$} \{0,1\}^\lambda$
if $State[\psi(q_j)][w_{j,2}] = null$ **then**
  $State[\psi(q_j)][w_{j,2}] \xleftarrow{\$} \{0,1\}^\lambda$
return $(State, State[\psi(q_j)])$

B. Security Proof

To demonstrate the JXTMM is adaptive $\mathcal{L}$-secure and volume-hiding under the leakage function, we proceed the security proof with two theorems as described below:

Theorem 1. *If the F is a secure pseudorandom function and (Enc, Dec) is a standard IND-CPA secure symmetric-key encryption scheme, the JXTMM scheme is $\mathcal{L}$-semantically-secure against adaptive attacks, where $\mathcal{L} = (dsize, (qeq, mrlen, jd, ip))$ is the leakage function defined above.*

Proof. To prove JXTMM is adaptively secure, we construct a simulator $\mathcal{S}$ consisting of SimSetup(1^λ, $dsize$) and SimSearch(1^λ, qeq, $mrlen$, jd, ip) as shown in Algorithm 3. With the simulator $\mathcal{S}$, we need to further demonstrate the probability of output '1' in the real game $\mathrm{Real}_{\mathcal{A}}^{\sum}(\lambda)$ is negligibly different from that in the ideal game $Ideal_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\sum}(\lambda)$ to any adversary $\mathcal{A}$. Here, $\sum$ is the JXTMM scheme. In the following, 8 games are described in details to formally prove the computational indistinguishability between the real and ideals games:

Game G_0: The first game G_0 is identical with the real JXTMM $\mathrm{Real}_{\mathcal{A}}^{\sum}(\lambda)$, since G_0 straightforwardly performs our proposed scheme. Thus we have:

$$\Pr[G_0 = 1] = \Pr[\mathrm{Real}_{\mathcal{A}}^{\sum}(\lambda) = 1]. \tag{9}$$

Game G_1: G_1 is almost the same to G_0, except the evaluations of $F_p(K_Z, \cdot)$, $F_p(K_{Z'}, \cdot)$, $F_p(K_I, \cdot)$, $F_p(K_W, \cdot)$ are substituted by outputs from distinct random functions or chosen from the relevant domain and range.

The distinguishing advantage between G_0 and G_1 is equal to that of PRF against an adversary with at most N calls to F. Therefore, there exists an efficient adversary $\mathcal{B}_1$ to achieve the advantage, then we have:

$$|\Pr[G_1 = 1] - \Pr[G_0 = 1]| \leq 4 \cdot Adv_{F,\mathcal{B}_1}^{\mathrm{PRF}}(\lambda). \tag{10}$$

Game G_2: G_2 is identical to G_1, except the ciphertexts e and $xtag$ are encrypted with a random value $\{0,1\}^\lambda$ to replace the actual identifier used in the real game. Let the number of encrypt steps be polynomial$poly(\lambda)$, the advantage of adversary $\mathcal{B}_2$ can be formalized as:

$$|\Pr[G_2 = 1] - \Pr[G_1 = 1]| \leq poly(\lambda) \cdot Adv_{\sum,\mathcal{B}_2}^{IND-CPA}(\lambda). \tag{11}$$

Game G_3: G_3 is nearly the same as the prior G_2, with the exception the blind values y, y' are randomly chosen from Z_p^*. Since the changes introduced in G_1, a random function $F_p(\cdot,\cdot)$ is selected to determine the parameters z, zz, z_{cnt}, zz_{cnt} are uniformly and independently distributed. Thus, the values of y, y' are also uniform and independent, since $y = xind - (z + z_{cnt})$ and $y' = xind - (zz + zz_{cnt})$. Finally, substituting y, y' with random values will not alter the distribution of the resulted game, then we can conclude that:

$$\Pr[G_3 = 1] = \Pr[G_2 = 1]. \tag{12}$$

Game G_4: G_4 is almost identical to the game G_3, with the exception it replaces the hash functions $h_0(\cdot)$ and $h_1(K_H, \cdot)$ with random function. Based on the security of hash function, we have:

$$\Pr[G_4 = 1] = \Pr[G_3 = 1]. \tag{13}$$

Game G_5: G_5 is distributionally identical to G_4 except it replaces the encryption function Enc in XOR filter with a random function. By the secure characteristics of AES, G_5 and G_4 are computed indistinguishable. Thus, we have:

$$\Pr[G_5 = 1] = \Pr[G_4 = 1]. \tag{14}$$

Game G_6: G_6 is exactly like G_5, except the outputs of the symmetric encryption on retrievals are replaced by random strings. Due to the same distribution outputs, G_5 and G_6 are distinguishable with negligible probability, i.e.,

$$\Pr[G_6 = 1] - \Pr[G_5 = 1] \leq Adv_{\mathcal{E},\mathcal{B}_3}^{IND-CPA}(\lambda). \tag{15}$$

Game G_7: In G_7, instead of using the cPRF function directly, it chooses a randomized function. According to the secure characteristics of cPRF, G_6 and G_7 are indistinguishable, i.e.,

$$\Pr[G_7 = 1] - \Pr[G_6 = 1] \leq \cdot Adv_{F,\mathcal{B}_1}^{\mathrm{PRF}}(\lambda). \tag{16}$$

It is significant to mention that the simulator $\mathcal{S}$ takes (1^λ and the leakage function $\mathcal{L} = (dsize, (qeq, mrlen, jd, ip))$ as

inputs and outputs a simulated EXMM along with a simulated state list $State$ to the adversary $\mathcal{A}$. Since the process of building EXMM are identical to those in G_7, we can come to the conclusion that,

$$\Pr[Ideal^{\sum}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)=1]=\Pr[G_7=1]. \tag{17}$$

Finally, the Theorem 1 is proved.

□

Theorem 2. *The leakage function $\mathcal{L}=(dsize,\ (qeq,\ mrlen,\ jd,\ ip))$ for the JXTMM scheme is volume-hiding.*

Proof. To formalize the volume-hiding, we give its definition through a leakage function as follows:

Definition 3. *A leakage function $\mathcal{L}=(\mathcal{L}_{Setup},\ \mathcal{L}_{Search})$ is volume-hiding, if and only if, for any adversary $\mathcal{A}$ and $1\leq \ell_{\max}\leq n$,*

$$p_0^{\mathcal{A},\mathcal{L}}(n,\ell_{\max})=p_1^{\mathcal{A},\mathcal{L}}(n,\ell_{\max}) \tag{18}$$

where, $\theta\in\{0,1\}$, and $p_\theta^{\mathcal{A},\mathcal{L}}(n,\ell_{\max})$ denotes the probability of $\mathcal{A}$ outputs '1' in a game $G_\theta^{\mathcal{A},\mathcal{L}}(n,\ell_{\max})$.

Details of the game $G_\theta^{\mathcal{A},\mathcal{L}}(n,\ell_{\max})$ are as follows:

(1) Given a total of n attribute-value pairs and a maximum volume length $\ell_{\max}$, the adversary $\mathcal{A}$ is capable of generating two distinct signatures: $\mathcal{S}_0=\{(w_i,\ell_0)(w_i))\}_{i\in[n]}$ and $\mathcal{S}_1=\{(w_i,\ell_1)(w_i))\}_{i\in[n]}$, where both signatures share the same n and $\ell_{\max}$.

(2) With these two signatures $\mathcal{S}_0$ and $\mathcal{S}_1$, a challenger $\mathcal{C}$ randomly chooses one of them, denoted as$\mathcal{S}_\theta$. This is to construct a multi-map MM_θ composed of randomly chosen attribute-value and identifiers. Thus, the multi-map MM_θ is then encrypted, and $\mathcal{C}$ proceeds to send the encrypted setup information $\mathcal{L}_{Setup}$ to $\mathcal{A}$.

(3) The adversary $\mathcal{A}$ selects two distinct join attributes $\psi(w_1)$, $\psi(w_2)$ for the quires. In response, the $\mathcal{C}$ returns the search-related information $\mathcal{L}_{Search}$ to $\mathcal{A}$.

(4) Finally, $\mathcal{A}$ outputs a bit $b\in\{0,\ 1\}$.

At last, the proof is given as follows:

- The leakage function is given as $\mathcal{L}=(\mathcal{L}_{Setup},\ \mathcal{L}_{Search})=(dsize,\ (qeq,\ mrlen,\ jd,\ ip))$.
- The adversary $\mathcal{A}$ generates two multi-map signatures $\mathcal{S}_0$, $\mathcal{S}_1$, each with size n and a maximum volume length $\ell_{\max}$, and presents them to the challenger $\mathcal{C}$.
- $\mathcal{C}$ then randomly selects one of these signatures to construct both an encrypted multi-map and a hashed multi-map, initiating an interaction with the adversary $\mathcal{A}$.
- Then for JXTMM, the setup leakage $\mathcal{L}_{Setup}$ includes the data size $dsize(\mathrm{MM})=n$, while the search leakage $\mathcal{L}_{Search}$ includes the maximum volume length $mrlen(\mathrm{MM})=\ell_{\max}$. The leakage functions of $\mathcal{L}_{Setup}$ and $\mathcal{L}_{Search}$ are identical for both signatures, due to the following reasons:
  (1) It deserves to be noted that the the query equation qeq is independent of the multi-map construction and that the leakage of data size $dsize$ and maximum volume length $mrlen$ is the same for both multi-map signatures.
  (2) Moreover, the ip leakage leaks only to the challenger, not to the adversary; the jd leakage is essentially query-invariant, thus it is considered benign.
- As a result, the adversary $\mathcal{A}$ cannot distinguish the difference between any two signatures, i.e. $p_0^{\mathcal{A},\mathcal{L}}(n,\ell_{\max})=p_1^{\mathcal{A},\mathcal{L}}(n,\ell_{\max})$. Therefore, it can be inferred that JXTMM scheme is capable to hide volume.

□

VII. Complexity Analysis

In the following, we theoretically compare our scheme JXTMM with the single keyword volume-hiding scheme (i.e. XORMM [17]), the standard conjunctive queries scheme (i.e. OXT [12]) and the table join queries scheme (i.e. JXT [13], TNT-QJ [14], OTJXT [15], FBJXT [16]) concerning storage cost, computational and communication overhead.

For simplicity, we consider a database including two tables, Tab_1 and Tab_2, each with n multi-maps consisting of attribute-value pairs and corresponding identifiers, m records, and T join attributes. Assume a query involving two attribute-value pairs, w_1 and w_2, which is performed on the join of Tab_1 and Tab_2 with based on the attribute $attr^*$. Table I provides a comparison summary.

A. Storage Cost

To start, we analyze the storage costs for XORMM, OXT, JXT, TNT-QJ, OTJXT, FBJXT and JXTMM. For JXT, TNT-QJ, OTJXT, FBJXT and JXTMM, when performing table join queries, the TSet (EMM) and XSet (XMM) structures are built separately for each table. Therefore, the overall cost of storage for these queries is basically identical the combined cost of storing storage each table independently.

Specifically, based on the construction of a XOR filter [30], XORMM maps a multi-map of size n to the XOR filter by encrypting and padding dummy values. Therefore, the storage cost of the XORMM scheme is $1.23n{\cdot}ev+\beta$.

OXT generates the TSet and XSet to form an encrypted database. The number of entries stored in TSet and XSet corresponds to the number of attribute-value pairs and their identifiers. Specifically, the encrypted value ev and the blind value y are stored in TSet, while the $xtag$ value xt is stored in XSet. Therefore, its storage cost is $(xt+ev+y)n$.

JXT is similar to OXT in that it produces TSet and XSet to form the encrypted database. The TSet consists of tuples containing n encrypted values ev and blind values y, while the XSet consists of $xtags$ that store all possible combinations of record identifiers and their associated join attribute-value pairs, with a storage size of $m{\cdot}T$. Thus, the total storage cost of JXT is $(ev+y)n+xt{\cdot}m{\cdot}T$. Similar to JXT, both TNT-QJ and FBJXT employ TSet and XSet structures to organize the encrypted database. Consequently, their storage costs can also be expressed as $(ev+y)n+xt{\cdot}m{\cdot}T$. In contrast, OTJXT adopts a novel data encoding strategy that eliminates the need for an XSet, relying solely on the TSet. As a result, its storage cost reduces to $(ev+y)n\cdot T$.

JXTMM generates the encrypted multi-maps, EMM and XMM. In particular, for each attribute-value pair in the table,

EMM maintains the encrypted identifiers and their blind values, while XMM stores the hash values of $xtags$. Both have a storage size proportional n. Therefore, the total storage size of JXTMM is $1.23n{\cdot}(ev + y + h) + \beta$.

B. Computational Overhead

The computational overhead involved in performing a search is now asymptotically analyzed. When executing a query for a single keyword, the number of matched results in the XORMM scheme is determined by the underlying data structure's query communication complexity. Therefore, both its client and server computational overheads primarily depend on the maximum volume length ($\ell_{\max}$).

In the context of a conjunctive query supporting multiple keywords using OXT, the client performs $O(1)$ computations to generate the search token for w_1, while the number of computations required to generated search tokens for the remaining keywords $(w_2, \cdots, w_x)$ is $x{\cdot}\ell_1$. Consequently, the client's computational overhead is $O(x{\cdot}\ell_1)$. The server's computation is divided into two steps. Initially, the search token generated for w_1 by the client is used to perform a TSet lookup, which has a computational complexity of $O(\ell_1)$. Subsequently, $O(x{\cdot}\ell_1)$ computations are performed to determine whether the search tokens generated for the remaining keywords belong to XSet. Therefore, the total computational complexity of the server is $O(\ell_1) + O(x{\cdot}\ell_1)$.

When performing a join search on two tables, Tab_1 and Tab_2. In JXT, the client generates search tokens (stag) for w_1 and w_2. This is achieved by executing the stag-generation algorithm for the TSet by using $O(1)$ invocations. The client also computes join tokens, with the number of such computations being $\ell_1 + \ell_2$. Hence, the computation overhead is $O(\ell_1 + \ell_2)$. On the server side, TSet lookups are performed for w_1 and w_2, with a computational complexity $\ell_1 + \ell_2$. Subsequently, the server calculates and verifies the candidate $xtag$ entries associated with each pair of join tokens, with a computational complexity $\ell_1\ell_2$. Therefore, the total computational overhead for the server is $O(\ell_1{+}\ell_2){+}O(\ell_1\ell_2)$. FBJXT is similar to JXT, where the client computation overhead is $O(\ell_1 + \ell_2)$, and the server computation overhead is $O(\ell_1 + \ell_2) + O(\ell_1 \cdot \ell_2)$. TNT-QJ, due to the meta-keyword transformation, incurs a client computation overhead of $O(x \cdot n^{'} + x \cdot \ell_{max})$. On the server side, TNT-QJ performs SFMST-based searches, which lead to a server computation overhead of $O(logn) + O(\ell_1 \cdot \ell_2)$. In contrast, OTJXT has the same client computation overhead as JXT, while on the server side the candidate computation during result retrieval is linear, yielding a complexity of $O(\ell_1 + \ell_2)$.

In JXTMM, the client makes $O(1)$ calls to the $F.Cons$ algorithm of cPRF to compute the search tokens (tok) for w_1 and w_2. The client then computes the join tokens. This step repeats $\ell_{\max_1}$ and $\ell_{\max_2}$ times, respectively. After that, the client performs decryption and membership checks $\ell_{\max_1} \cdot \ell_{\max_2}$ times. Thus, the client's total computational overhead is $O(\ell_{\max_1} + \ell_{\max_2} + \ell_{\max_1} \cdot \ell_{\max_2})$. The server's computation consists of two parts. First, it uses the client's search tokens to retrieve the encrypted result sets for w_1 and w_2.This requires $O(\ell_{\max_1} + \ell_{\max_2})$ computations. Second, it computes the hash values of the $xtag$ entries corresponding to the join tokens and stores them in the bloom filter. This requires $O(\ell_{\max_1} \cdot \ell_{\max_2})$operations. Therefore, the server's total computational overhead is $O(\ell_{\max_1} + \ell_{\max_2}) + O(\ell_{\max_1} \cdot \ell_{\max_2})$.

C. Communication Overhead

Next, we analyze the communication overhead involved in executing a search.

For a single keyword query, the client sends only a token to the server, resulting in a communication complexity of $O(1)$. The server responses with an encrypted result set of size $O(\ell_{\max})$, making the total communication complexity $O(\ell_{\max})$.

In the case of conjunctive queries involving multiple keywords using OXT, the client sends $O(x{\cdot}\ell_1)$ terms to the server, and the server responds with $O(x{\cdot}\ell_1)$ terms. Consequently, the total communication complexity is $O(x{\cdot}\ell_1)$. When executing a join search on two tables, Tab_1 and Tab_2.

In JXT, the client sends $O(\ell_1 + \ell_2)$ terms to the server, while the server's response contains $O(|DB(q)|)$ terms. Therefore, the overall communication complexity is $O(\ell_1 + \ell_2) + O(|DB(q)|)$. For TNT-QJ, the client uploads $O(x{\cdot}n^{'})$ tokens, and the server returns $O(\ell_1{+}\ell_2)$ results, leading to an overall communication complexity of $O(x \cdot n^{'}){+}O(\ell_1{+}\ell_2)$. For FB-JXT, both the client and the server exchange $O(\ell_1{+}\ell_2)$ terms, resulting in a total communication complexity of $O(\ell_1{+}\ell_2)$. In the case of OTJXT, the client sends $O(\ell_1{+}\ell_2)$ join tokens, and the server replies with $O(|DB(q)|)$ encrypted results, yielding a total communication complexity of $O(\ell_1{+}\ell_2){+}O(|DB(q)|)$.

In JXTMM, the client sends the search tokens and join tokens to the server, with a total size of $O(\ell_{\max_1} + \ell_{\max_2})$. The server returns the candidate value set of size $O(\ell_{\max_1} \cdot \ell_{\max_2})$, along with the bloom filter of size O(BF).Therefore, the overall communication complexity is $O(\ell_{\max_1} + \ell_{\max_2}) + O(\ell_{\max_1} \cdot \ell_{\max_2}) + O$(BF).

VIII. Performance Evaluation

In this section, we compare the more fundamental and representative join query JXT [13] scheme with our proposed JXTMM scheme to access its performance. We do not consider XORMM and OXT in this section since their search type are single keyword and conjunctive search, do not support join table queries.

The setup of the two experiments is first described, and then the overhead of the setup and query phases are shown by the simulation results.

A. Experiment Setup

We conduct our experiments on a machine equipped with an Intel i7-13700 @ 5.0GHz and 16GB of RAM. Our implementation of the proposed JXTMM scheme, along with the baseline JXT scheme for comparison, is developed in Java. Cryptographic operations, including PRFs and hash functions, are performed using JDK libraries such as AES and SHA-256.

We use the TPC-H benchmark dataset in our experiments. This dataset is widely adopted for testing database systems under complex analytical queries and large-scale data. Thus,

following common practice in encrypted database research [46], we evaluate on the TPC-H benchmark with the customer and orders tables with scale factors of 0.1, 0.2, 0.3, 0.4, and 0.6. The CUSTOMER and ORDER tables consist of 8 and 9 attributes, respectively, and both include the join attribute *custkey*. Under these settings, the CUSTOMER table contains 15,000, 30,000, 45,000, 60,000, and 90,000 rows, respectively, while the ORDER table contains 150,000, 300,000, 450,000, 600,000, and 900,000 rows. We extract only a subset of columns to reduce irrelevant complexity while keeping the attributes needed for evaluation. For tables with a scale factor of 0.01, we add a synthetic attribute to control search selectivity. Specifically, 1% of the rows are assigned value 1, 2% value 2, 4% value 3, and 8% value 4. This setup allows us to simulate different filtering levels and evaluate the proposed scheme under varying selectivity.

Specifically, the TSet in JXT scheme is constructed using an inverted index, while the XSet in the JXT scheme is created as a dynamic array. Finally, we implement the cPRF by constructing a GGM tree. Based on the parameter settings in XORMM [17], we set $\beta = 8$ to improve the success probability of the dissimilarity filter and required storage.

We present a comprehensive analysis of the setup and search overhead in JXT and JXTMM. In particular, we begin by assessing the storage size and the time required to create encrypted tables across different data sizes. Subsequently, to access the effect of the resultant volume, we compare the search time costs of the two schemes across different correlation volume lengths.

B. Evaluation and Comparison

Storage Cost. Each table is stored separately in both JXT and JXTMM. The total storage overhead equals the sum of all encrypted tables. Thus, the key metric is the overhead for each table. In JXT, the overhead comes from TSet and XSet, whereas in JXTMM, it comes from EMM and XMM. The main difference lies in construction: JXTMM uses XOR filters, while JXT builds TSet with an inverted index and XSet with a dynamic array. For a multi-map of size n, JXTMM generates an EMM of size $1.23n + \beta$. As a result, JXTMM consumes more storage than JXT, as shown in Fig. 5. However, this additional cost enables both volume-hiding and result-hiding. The trade-off is reasonable since the overall performance is not significantly affected.

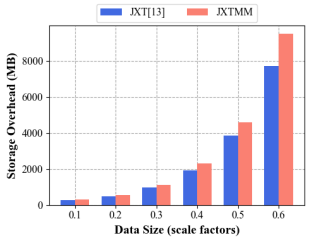


Fig. 5: Storage overhead.

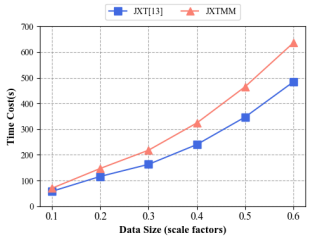


Fig. 6: Setup time.

Setup Evaluation. We next evaluate the setup efficiency of table join queries under different data sizes. In JXTMM, ciphertexts are generated using multi-map mapping with XOR filters, while those in JXT are generated with AES. As shown in Fig. 6, JXTMM scheme incurs slightly higher overhead. In practice, setup is performed only once, and the extra cost remains within an acceptable range. Thus, the setup efficiency of JXTMM is practical.

Search Evaluation. We analyze the search efficiency across different dataset sizes and search selectivity.

(1) Effect of data size. We first evaluate the time cost of table join queries for both JXTMM and JXT across different dataset sizes. For each dataset, we query 30 randomly selected keywords. As shown in Fig. 7, the search time of both JXT and JXTMM increases with the data size. This is because larger datasets increase both keyword volume (ℓ_1,ℓ_2) and maximum volume (ℓ_{max}). However, JXTMM exhibits a narrower fluctuation range: its performance depends only on ℓ_{max}, whereas JXT depends on varying (ℓ_1,ℓ_2). This makes JXTMM more stable and scalable in large-scale encrypted databases. The stability demonstrates the robustness of JXTMM against growing data. Since JXTMM consistently shows stable search times regardless of dataset size, it provides more practical efficiency in large-scale encrypted databases.

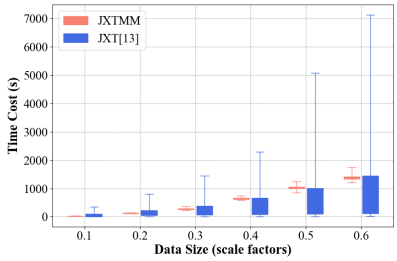


Fig. 7: Search time with data size.

(2) Effect of selectivity. Next, we query keywords with varying search selectivities to further analyze the search cost. We measured the time cost in each phase: token generation (search and join tokens), search execution, and result decryption.

Specifically, Fig. 8a compares the time to generate tokens for the JXTMM and JXT schemes with varying data sizes. The token generation time of both schemes remains largely unaffected by search selectivity. For a more detailed comparison, Table III reports the time required for generating search and join tokens under the two schemes. In search token generation, JXTMM achieves slightly faster performance than JXT because it employs a cPRF function. In join token generation, the time cost of JXTMM depends only on ℓ_{max}. Although the token generation time of JXT is theoretically related to ℓ_1 and ℓ_2, the client cannot determine their exact values during the process and must also rely on ℓ_{max}. As a result, the join token generation time remains unchanged for both schemes.

TABLE III: Generate Token Time Cost (ms).

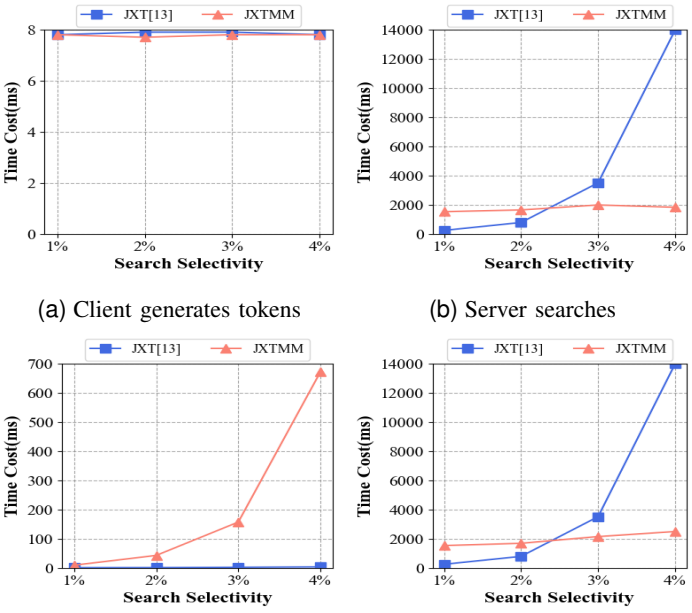
Search Selectivity	Search Token		Join Token	
	JXT	JXTMM	JXT	JXTMM
1%	0.034	0.016	7.74	7.91
2%	0.033	0.016	7.90	7.81
4%	0.033	0.016	7.81	7.99
8%	0.034	0.016	7.83	7.81

Fig. 8b illustrates the comparison of time costs during the server search phase, revealing that the time cost of JXT varies with search selectivity, i.e., the volume of queried keywords, while that of JXTMM remains nearly constant. Moreover,

once search selectivity exceeds a certain threshold, JXTMM achieves higher server-side query efficiency than JXT, owing to its use of bloom filters to optimize search performance.

Fig. 8c compares the client decryption times for both schemes. The client decryption times of both JXT and JXTMM increases with search selectivity. In JXTMM, the verification time required to check whether the result sets satisfies the query conditions is also affected by search selectivity. Moreover, JXTMM incurs higher decryption costs than JXT because it requires additional verification during decryption to ensure that the result sets satisfies the query conditions.

Fig. 8d shows the total execution time of join queries. The cost of JXT grows rapidly as selectivity increases, whereas that of JXTMM grows more slowly. Once search selectivity reaches a certain level, the search time of JXT is exceeds that of JXTMM. This stability further demonstrates JXTMM's robustness, as its performance is less affected by increasing selectivity. In addition, JXTMM not only maintains consistent search performance as selectivity increases but also provides a clear advantage in real-world applications, where high selectivity occurs frequently.

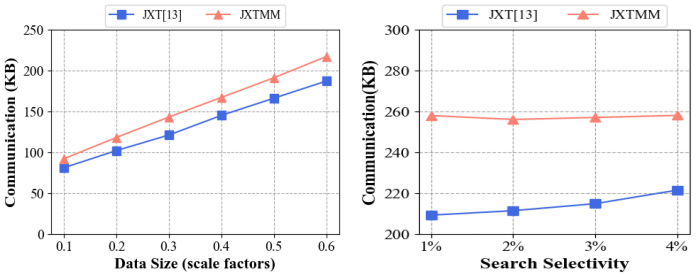


(a) Client generates tokens (b) Server searches

(c) Client decrypts (d) Total search

Fig. 8: Comparison of the search time cost with different search selectivity.

(3) Detailed max selectivity test. To further evaluate the impact of maximum selectivity on JXTMM search time, we configured each table with a maximum search selectivity of 8%, 16%, 32%, and 64%, while keeping the corresponding value fixed at 1. The results, shown in Fig. 9, indicate that the query time of JXTMM increases with higher maximum selectivity, i.e., ℓ_{max}. As ℓ_{max} grows, more dummy entries are introduced, requiring both the client and the server to process and decrypt a larger number of matching entries.



(a) Bandwidth with data size. (b) Bandwidth with search selectivity.

Fig. 11: Communication bandwidth.

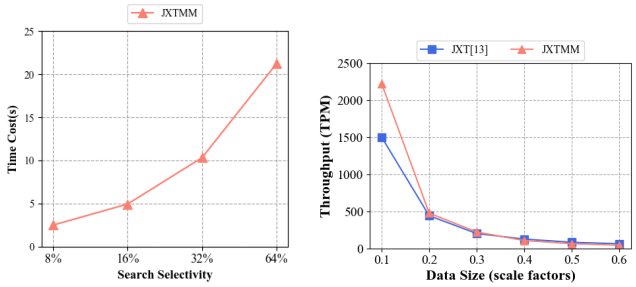


Fig. 9: Search time with maximum search selectivity. Fig. 10: Throughput with data size.

Network Condition Evaluation. (1) Throughput. We evaluated the query throughput of JXTMM and JXT across different data scales, measured in transactions per minute (TPM). The experimental conditions were kept consistent, with each dataset size using 30 randomly selected keywords. As shown in the Fig 10, the throughput of both schemes decreased as the data scale increased. This is because larger data scales lead to higher values of ℓ_{max}, ℓ_1 and ℓ_2, which in turn require longer search times for each query, thereby reducing the total number of queries processed per minute. Notably, JXTMM outperformed JXT in terms of throughput on smaller datasets. This suggests that JXTMM is more efficient when the number of query results is relatively small. Although, as the database size increases, the query tasks are generally dominated by data volume, causing the throughput of both schemes to converge, the initial advantage of JXTMM highlights its efficiency. This demonstrates that in environments where the query result scale is not excessively large, JXTMM exhibits high responsiveness and delivers superior practical performance.

(2) Communication Cost. We evaluated the communication overhead of JXT and JXTMM under varying data sizes and search selectivities. As illustrated in Fig. 11, JXT achieves higher communication efficiency in both cases. This is because, in JXTMM, bloom filters are employed to securely store join conditions, and dummy values are introduced during transmission to prevent result leakage. These steps increase communication cost but are necessary for security. As shown in Fig. 11a, communication overhead grows with data size due to increases in ℓ_{max}, ℓ_1 and ℓ_2. In contrast, Fig. 11b shows that JXT's overhead rises steadily with search selectivity, while JXTMM remains almost constant, since its communication depends only on ℓ_{max}, which is fixed in this experiment. Although JXTMM incurs slightly higher communication cost, its stability across varying selectivities and data sizes makes

it more practical for real-world applications.

IX. Limitations and Future Work

In this section, we discuss the limitations of JXTMM as well as the research directions worth further exploration. We summarize some of these below.

Lack of Dynamic Support. JXTMM is designed for static databases and does not support dynamic operations such as insertion, deletion, or modification of records after the initial setup. This static assumption is common in many SSE schemes [14], [15] as it simplifies design construction and reduces overall complexity. However, it inevitably limits its applicability in practical scenarios where datasets are frequently updated. The limitation mainly arises from the use of the XOR filter [30] as the core data structure: because each element is represented across multiple positions, efficient insertions and deletions are inherently difficult. Future work may explore integrating dynamic data structures into the encrypted multi-map framework, such as dynamic cuckoo hashing [47], oblivious key-value stores [48], or random bucket techniques [49]. These structures could be adapted to enable secure incremental updates while preserving strong security guarantees. Developing a dynamic variant of JXTMM is therefore an important direction for future research.

Applicability in Multi-User Environments. JXTMM currently supports only a single-user setting. In this model, a single client generates search tokens and interacts with the server. However, many applications in practice require multiple authorized users to access the same encrypted database. Recently, Li et al. [16] proposed a dynamic and secure join query protocol for multi-user environments. Wang et al. [50] presented SMKSE, which reduces pattern leakage in such settings. These techniques such as key-homomorphic PRFs [16] or SMKSE's blinding-factor and garbled bloom filter [50]could be integrated into JXTMM to enable controlled multi-user access. Extending JXTMM in this direction is an important task for future research.

Concurrency Query Execution. Due to the use of a bloom filter (BF) to encode candidate join pairs tag during the server-side search procedure, the bit array must be updated based on query-specific tokens and cross-tags. As the BF is tightly coupled to the in-flight query, simultaneous updates from multiple queries or threads would interfere with each other, leading to cross-query contamination and unverifiable membership checks. Future work could enable safe concurrency by isolating per-query state, for example by assigning each query a disjoint namespace to prevent cross-talk. Another direction is to explore bloom filter variants that naturally support parallelism, such as partitioned bloom filters [51], which divide the bit array into disjoint segments and assign each hash function to a separate partition, thereby simplifying concurrent updates. We leave the design of a concurrency pattern-hiding join protocol for JXTMM to future work.

X. Conclusion

In this paper, we investigate secure table join queries in encrypted relational databases without requiring join precomputation. We propose an efficient volume-hiding and result-hiding JXTMM scheme to support table join queries. Building on table join queries, our scheme mitigates leakage that may lead to volume and result attacks. Moreover, JXTMM improves the efficiency of table join queries in certain aspects.

However, JXTMM is applicable only to static databases, and the design of dynamic volume-hiding EMMs that support table join queries remains a challenging problem.

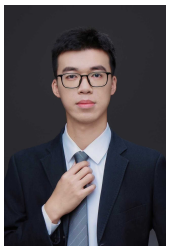
References

[1] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *Proc. ASIACRYPT*, ser. LNCS, 2010, pp. 577–594.
[2] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proc. ACM CCS*, 2006, pp. 79–88.
[3] Y. Wang, S.-F. Sun, J. Wang *et al.*, "Achieving searchable encryption scheme with search pattern hidden," *IEEE Transactions on Services Computing*, vol. 15, no. 2, pp. 1012–1025, March-April 2022.
[4] Y. Zheng, P. Xu, M. Wang *et al.*, "Themis: Robust and light-client dynamic searchable symmetric encryption," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 8802–8816, 2024.
[5] K. Kurosawa and Y. Ohtaki, "Uc-secure searchable symmetric encryption," in *Proc. FC*, ser. LNCS, 2012, pp. 285–298.
[6] Z. Zhang, J. Wang, Y. Wang *et al.*, "Towards efficient verifiable forward secure searchable symmetric encryption," in *Proc. ESORICS*, ser. LNCS, 2019, pp. 215–234.
[7] D. Cash, S. Jarecki, C. S. Jutla *et al.*, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Proc. CRYPTO*, 2013, pp. 353–373.
[8] D. Cash, J. Jaeger, S. Jarecki *et al.*, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *Proc. NDSS*, 2014.
[9] S. Faber, S. Jarecki, H. Krawczyk *et al.*, "Rich queries on encrypted data: Beyond exact matches," in *Proc. ESORICS*, 2015, pp. 123–145.
[10] S. Patranabis and D. Mukhopadhyay, "Forward and backward private conjunctive searchable symmetric encryption," in *Proc. NDSS*, 2021, pp. 23 116–23 130.
[11] S. Kamara and T. Moataz, "Sql on structurally-encrypted databases," in *Proc. ASIACRYPT*, 2018, pp. 149–180.
[12] D. Cash, R. Ng, and A. Rivkin, "Improved structured encryption for sql databases via hybrid indexing," in *Proc. ACNS*, 2021, pp. 480–510.
[13] C. Jutla and S. Patranabis, "Efficient searchable symmetric encryption for join queries," in *Proc. ASIACRYPT*, ser. LNCS, 2022, pp. 304–333.
[14] J. Wu, K. Zhang, L. Wei *et al.*, "Practical searchable symmetric encryption for arbitrary boolean query-join in cloud storage," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 10 086–10 098, 2024.
[15] Q. Xu, J. Wang, S.-F. Sun *et al.*, "Practical equi-join over encrypted database with reduced leakage," *IEEE Transactions on Knowledge and Data Engineering*, vol. 37, no. 5, pp. 2846–2860, 2025.
[16] H. Li, D. He, Q. Feng *et al.*, "A dynamic and secure join query protocol for multi-user environment in cloud computing," *IEEE Transactions on Cloud Computing*, vol. 13, no. 2, pp. 512–525, 2025.
[17] J. Wang, S.-F. Sun, T. Li *et al.*, "Practical volume-hiding encrypted multi-maps with optimal overhead and beyond," in *Proc. ACM CCS*, 2022, pp. 2825–2839.
[18] S. Kamara, T. Moataz, and O. Ohrimenko, "Structured encryption and leakage suppression," in *Proc. CRYPTO*, 2018, pp. 339–370.
[19] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: ramification, attack and mitigation," in *Proc. NDSS*, 2012, pp. 12–26.
[20] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic attacks on secure outsourced databases," in *Proc. ACM CCS*, 2016, pp. 1329–1340.
[21] P. Grubbs, M. Lacharite, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proc. ACM CCS*, 2018, pp. 315–331.
[22] Q. Song, Z. Liu, J. Cao *et al.*, "Sap-sse: Protecting search patterns and access patterns in searchable symmetric encryption," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1795–1809, 2021.
[23] S. Kamara and T. Moataz, "Computationally volume-hiding structured encryption," in *Proc. EUROCRYPT*, ser. LNCS, 2019, pp. 183–213.
[24] S. Patel, G. Persiano, K. Yeo, and M. Yung, "Mitigating leakage in secure cloud-hosted data structures: volume-hiding for multi-maps via hashing," in *Proc. ACM CCS*, 2019, pp. 79–93.

[25] J. Li, L. Ji, Y. Zhang, *et al.*, "Response-hiding and volume-hiding verifiable searchable encryption with conjunctive keyword search," *IEEE Transactions on Computers*, vol. 74, no. 2, pp. 455–467, Feb. 2025.
[26] H. Wang, K. Fan, C. Yu, *et al.*, "Beyond access pattern: Efficient volume-hiding multi-range queries over outsourced data services," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 2509–2522, 2025.
[27] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "Seal: Attack mitigation for encrypted databases via adjustable leakage," in *Proc. USENIX Security*, 2020, pp. 2433–2450.
[28] J. Ning, X. Huang, G. S. Poh *et al.*, "Leap: Leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset," in *Proc. ACM CCS*, 2021, pp. 2307–2320.
[29] Z. Gui, O. Johnson, and B. Warinschi, "Encrypted databases: new volume attacks against range queries," in *Proc. ACM CCS*, 2019, pp. 361–378.
[30] T. M. Graf and D. Lemire, "Xor filters: faster and smaller than bloom and cuckoo filters," *ACM Journal of Experimental Algorithmics*, vol. 25, p. 16, Mar. 2020.
[31] X. Meng, S. Kamara, K. Nissim, and G. Kollios, "Grecs: graph encryption for approximate shortest distance queries," in *Proc. ACM Conf. CCS*, 2015, pp. 504–517.
[32] S. Kamara, "Restructuring the nsa metadata program," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, 2014, pp. 235–247.
[33] Y. Zhang, A. O'Neill, M. Sherr, and W. Zhou, "Privacy-preserving network provenance," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1550–1561, 2017.
[34] B. Chen, T. Xiang, D. He *et al.*, "Bpvse: Publicly verifiable searchable encryption for cloud-assisted electronic health records," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 3171–3184, 2023.
[35] D. X. Song, D. A. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE Symp. S&P*, May 2000, pp. 44–55.
[36] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *Proc. ESORICS*, ser. LNCS, 2019, pp. 283–303.
[37] G. Amjad, S. Kamara, and T. Moataz, "Injection-secure structured and searchable symmetric encryption," in *Proc. ASIACRYPT*, ser. LNCS, 2023, pp. 232–262.
[38] F. Liu *et al.*, "Volume-hiding range searchable symmetric encryption for large-scale datasets," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 3597–3609, Jul-Aug 2024.
[39] L. Chen, Y. Xue, Y. Mu, L. Zeng, F. Rezaeibagha, and R. Deng, "Casesse: Context-aware semantically extensible searchable symmetric encryption for encrypted cloud data," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1011–1022, 2023.
[40] F. Li, J. Ma, Y. Miao, P. Wu, and X. Song, "Beyond volume pattern: Storage-efficient boolean searchable symmetric encryption with suppressed leakage," in *Proc. ESORICS*, ser. LNCS, 2023, pp. 126–146.
[41] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proc. ACM CCS*, 2015, pp. 668–679.
[42] S. Kamara, A. Kati, T. Moataz *et al.*, "Sok: Cryptanalysis of encrypted search with leaker – a framework for leakage attack evaluation on real-world data," in *Proc. EuroS&P*, 2022, pp. 90–108.
[43] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks," in *Proc. SP*, 2021, pp. 1502–1519.
[44] D. Boneh and B. Waters, "Constrained pseudorandom functions and their applications," in *Proc. ASIACRYPT*, ser. LNCS, 2013, pp. 280–300.
[45] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech, "A family of perfect hashing methods," *Comput. J.*, vol. 39, no. 6, pp. 547–554, 1996.
[46] M. Dreseler, M. Boissier, T. Rabl, and M. Uflacker, "Quantifying TPC-H choke points and their optimizations," in *Proceedings of the VLDB Endowment*, vol. 13, no. 8. VLDB Endowment, Apr. 2020, pp. 1206–1220.
[47] F. Zhang, H. Chen, H. Jin, and P. Reviriego, "The logarithmic dynamic cuckoo filter," in *Proc. ICDE*, 2021, pp. 948–959.
[48] A. Bienstock, S. Patel, J. Y. Seo, and K. Yeo, "Near-optimal oblivious key-value stores for efficient psi, psu and volume-hiding multi-maps," in *Proc. USENIX Security*, 2023, pp. 301–318.
[49] S. Han, V. Chakraborty, M. T. Goodrich *et al.*, "Veil: A storage and communication efficient volume-hiding algorithm," *Proc. ACM Manag. Data*, vol. 1, no. 4, 2023.
[50] M. Wang, Z. Chen, Y. Miao *et al.*, "Cross-user leakage mitigation for authorized multi-user encrypted data sharing," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 1213–1226, 2024.
[51] P. S. Almeida, "A case for partitioned bloom filters," *IEEE Transactions on Computers*, vol. 72, no. 6, pp. 1681–1691, 2023.

Jinguo Li (Member, IEEE) received the B.S. degree in information security and the Ph.D. degree in computer science and technology from Hunan University, Changsha, China, in 2007 and 2014, respectively. He is currently a Professor with the College of Computer Science and Technology, Shanghai University of Electric Power, Shanghai, China. His research interests include information security and privacy, applied cryptography, and Trustworthy Artificial Intelligence.

Delong Cui received the bachelor's degree from Shandong University of Science and Technology, China, in 2022. She is currently pursuing her master's degree in the College of Computer Science and Technology at Shanghai University of Electric Power, China. Her research interests include cloud security and searchable encryption.



Junqin Huang is currently a research assistant professor at the School of Computer Science, Shanghai Jiao Tong University. He received the Ph.D. Degree in computer technology from Shanghai Jiao Tong University in 2024 and the B.Eng. degree in computer science and technology from University of Electronic Science and Technology of China in 2018. He serves as a member of the Youth Editorial Board for BCRA and Blockchain, a Lead Guest Editor for Electronics, and a TPC member for several international conferences, including IEEE Blockchain, IEEE ISPA, IEEE VTC. His research interests include blockchain, internet of things, data security, trusted computing.

Linghe Kong (Senior Member, IEEE) received the B.Eng. degree in automation from Xidian University in 2005, the master's degree in telecommunication from Telecom SudParis in 2007, and the Ph.D. degree in computer science from Shanghai Jiao Tong University in 2013. He is currently a Professor with the the School of Computer Science, Shanghai Jiao Tong University. Before that, he was a Post-Doctoral Researcher with Columbia University, McGill University, and the Singapore University of Technology and Design. His research interests include the Internet of Things, 5G, blockchain, and mobile computing.