Scalable Equi-Join Queries over Encrypted Database

Kai Du[*]
School of Cyber Engineering,
Xidian University,
Xi'an, China
kaidu@stu.xidian.edu.cn

Jianfeng Wang[*] [†]
School of Cyber Engineering,
Xidian University,
Xi'an, China
jfwang@xidian.edu.cn

Jiaojiao Wu[*]
School of Cyber Engineering,
Xidian University,
Xi'an, China
jiaojiaowujj@stu.xidian.edu.cn

Yunling Wang[*]
School of Cyberspace Security,
Xi'an University of Posts & Telecommunications,
Xi'an, China
ylwang@xupt.edu.cn

Abstract

Secure join queries over encrypted databases, the most expressive class of SQL queries, have attracted extensive attention recently. The state-of-the-art JXT (Jutla et al. ASIACRYPT 2022) enables join queries on encrypted relational databases without pre-computing all possible joins. However, JXT can merely support join queries over two tables (in encrypted databases) with some high-entropy join attributes.

In this paper, we propose an equi-join query protocol over two tables dubbed JXT+, that allows the join attributes with arbitrary names instead of JXT requiring the identical name for join attributes. JXT+ reduces the query complexity from $O(\ell_1 \cdot \ell_2)$ to $O(\ell_1)$ as compared to JXT, where ℓ_1 and ℓ_2 denote the numbers of matching records in two tables respectively. Furthermore, we present JXT++, the *first* equi-join queries across three or more tables over encrypted databases without pre-computation. Specifically, JXT++ supports joins of arbitrary attributes, i.e., all attributes (even low-entropy) can be candidates for join, while JXT requires high-entropy join attributes. In addition, JXT++ can alleviate sub-query leakage on three or more tables, which hides the leakage from the matching records of two-table join.

Finally, we implement and compare our proposed schemes with the state-of-the-art JXT. The experimental results demonstrate that both of our schemes are superior to JXT in search and storage costs. In particular, JXT+ (*resp.*, JXT++) brings a saving of 49% (*resp.*, 68%) in server storage cost and achieves a speedup of 51.7× (*resp.*, 54.3×) in search latency.

[*]All the authors contributed equally (Co-first authors).
[†]Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690377

CCS Concepts

• **Security and privacy → Management and querying of encrypted data**.

Keywords

Database Privacy; Encrypted Search; Structure Encryption; SQL Query

ACM Reference Format:
Kai Du, Jianfeng Wang, Jiaojiao Wu, and Yunling Wang. 2024. Scalable Equi-Join Queries over Encrypted Database. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3658644.3690377

1 Introduction

Encrypted search protocol enables the client to perform efficient search over encrypted databases. Symmetric searchable encryption (SSE), initiated by Song et al. [35], is a promising primitive for realizing efficient encrypted search, which has been extensively studied in the past two decades [9–13, 20, 21, 25, 32, 36, 37]. A long line of research has been conducted to achieve better trade-offs between security, functionality, and performance by revealing well-defined leakage information.

We note that, however, almost all SSE constructions support only keyword-based search on encrypted documents. As claimed in [25], most real-world data is usually stored and shared in relational databases. Roughly speaking, a relational database is a collection of tables with rows representing records and columns representing attributes, and most relational databases are equipped with the structured query language (SQL) for data querying and updating. A significant puzzle to be solved is how to perform expressive queries on encrypted relational database [25].

Hacigümüş et al. [18] first explicitly initiated the study of search on encrypted relational databases, where each attribute domain is split into a sequence of buckets and the associated bucket of the queried data will be returned. In 2011, Popa et al. [32] developed a SQL-aware encrypted database system called CryptDB, that can support a large class of SQL queries by assembling property-preserving encryption (PPE). Nevertheless, it has been witnessed in [29] that the PPE-style approach is vulnerable to leakage-abuse attacks.

Table 1: Comparison with prior secure join schemes.

Schemes	Pre-computation	Storage Cost	Query Computation		Multi-table Joins	Arbitrary Join Attribute
			Client Computation	Server Computation		
SPX [25]	●	$O(mn)+O(m^2T)$	$O(1)Prf + O(\|R\|)Dec$	$O(\ell_1 + \ell_2 + \|R_J\|)\mathsf{EMM_{qry}}^{\dagger}$	✓	✓
CNR [11]	◐	$O(mn)+O(mT)$	$O(1)Prf+O(\ell_1+\ell_2)Dec + O(\ell_1\ell_2)Join$	$O(\ell_1+\ell_2)(\mathsf{EMM_{qry}}+Prf)$	✗	✓
JXT [22]	○	$(2T+1)mn+mT$	$O(\ell_1+\ell_2)Prf+O(\|R\|)Dec$	$O(\ell_1 + \ell_2)\mathsf{EMM_{qry}} + O(\ell_1\ell_2)Xor$	✗	✗
JXT+	○	$3mnT$ (Worst)/ $2mnT+nT$(Best)	$O(\ell_1)Prf+O(\|R\|)Dec$	$O(\ell_1)(\mathsf{EMM_{qry}}+Xor) + O(\|R\|)H$	✗	✗
JXT++	○	$3.23mnT$(Worst)/ $1.23mnT+2nT$(Best)	$O(l_{max})Prf + O(\ell_1 l_{max})Dec$	$O(\ell_1)\mathsf{EMM_{qry}}+ O(\ell_1 l_{max})(Xor+H)$	✓	✓

Assume that the database consists of two tables, Tab_1 and Tab_2, and each has m rows, n columns, and T join attributes. We consider performing a query q = (Select $inds$ From Tab_1, Tab_2 Join On $attr_1^* = attr_2^*$ Where $w_1 \wedge w_2$). The symbols ●, ◐, and ○ denote fully, partially, and without pre-computation of all possible joins, respectively. Let $\ell_i = |\mathsf{DB}_{\mathsf{Tab}_i}(w_i)|$ and l_{max} be the maximum occurrence number of the combinations of each join-attribute value of $attr_1^*$ and all attribute-value pairs in Tab_1. R is the search result of the query q, and R_J is the search result of the join query q', where q' = (Select $inds$ From Tab_1, Tab_2 Join On $attr_1^* = attr_2^*$). Prf is the pseudorandom function operation, Dec is the decryption operation for symmetric encryption algorithm, H is the hash function, Xor is the exclusive-or operation. $\mathsf{EMM_{qry}}$ refers to the operation of retrieving an entry from the encrypted multi-map, and $Join$ refers to the equality test over plaintext.

†: If $R_J = \emptyset$, the server computation can be reduced to constant (i.e., $O(1)$).

To mitigate the leakage of join queries, Kamara et al. [25] introduced the first structured encryption scheme supporting join query named SPX, which relies crucially on pre-computing all possible joins and storing all the results in an encrypted version of the database. While it achieves optimal search complexity, SPX brings non-trivial storage overhead in some cases, especially when the database contains a large number of join attributes and the value distribution exhibits low entropy across all joinable attributes. That is, lower entropy across join attributes leads to a larger number of possible joins in pre-computation mode. Thus, SPX tends to configure high-entropy attributes as join attributes for storage efficiency. Subsequently, Cash et al. [11] presented a variant of SPX by introducing the technique of *partially pre-computed joins*, which can achieve less leakage and communication size at the expense of moderate client-side computation.

Recently, Jutla et al. [22] presented JXT, a novel join queries scheme without join pre-computation. Specifically, inspired by OXT [10], this protocol generates two new table-wise data structures TSet and XSet for each table, where TSet refers to an inverted index for attribute/value pairs to perform single-keyword search within a single table, and XSet contains all combinations of record identifier/join-attribute value pair. The basic idea is to retrieve the corresponding record identifiers matching the queried keywords for two tables, respectively. Later, the server computes all the cross-combinations of the matching record identifiers and join-attribute value pairs between two tables and checks whether the above combinations belong to the XSet. Thus, the query complexity is $O(\ell_1 \cdot \ell_2)$, where ℓ_1 and ℓ_2 denote the numbers of matching records in two tables. Although JXT can achieve two-table join queries without pre-computation, it requires the joinable attributes across tables with the same attribute name (i.e., natural join). In addition, JXT cannot be trivially extended to three or more tables due to the complication of generating join tokens on three or more tables.

In this work, we further investigate equi-join queries over encrypted databases without join pre-computation that can support low-entropy join attributes, even handling *three or more* tables. More precisely, it performs SQL queries in the following form:

$$\text{Select } inds \text{ From } tables \text{ Join On } (attr_1^* = \ldots = attr_k^*) \quad \text{Where } (w_1 \wedge \ldots \wedge w_k),$$

where $inds$ denotes record identifiers, $attr_i^*$ and w_i refer to the join attribute and the attribute-value pair in table Tab_i, respectively.

1.1 Our Contributions

In this paper, we make affirmative progress to secure join queries over encrypted relational databases without pre-computation. Specifically, we propose an efficient equi-join query scheme over two tables and further extend to scalable equi-join queries across three or more tables. Both of the constructions outperform the state-of-the-art JXT [22] in terms of query complexity and storage overhead. In addition, similar to JXT [22], our protocols can support flexible table addition by constructing index for newly added tables separately, while SPX [25] requires performing the setup process across all tables. Table 1 shows a brief comparison with prior works. More concretely, our main contributions can be summarized as follows:

- We propose JXT+, which supports equi-join queries over two tables without join pre-computation. Compared with the state-of-the-art JXT [22], JXT+ allows join attributes with arbitrary names, as opposed to the identical join attribute name in JXT. The query complexity of JXT+ is reduced from $O(\ell_1 \cdot \ell_2)$ to $O(\ell_1)$, where ℓ_1 and ℓ_2 are the numbers of matching records in the two queried tables. In addition, JXT+ can avoid the leakage information on join attributes by binding non-join-attribute value and join-attribute name in TSet.
- We present JXT++, to our best knowledge, the *first* equi-join queries over three or more tables without pre-computation. In particular, JXT++ can achieve joins of arbitrary attributes

even low-entropy attributes, eliminating the limitation of high-entropy join-attribute in JXT. Moreover, JXT++ can mitigate the sub-query leakage, stemming from multiple-table join, while surpassing JXT in query and storage efficiency.
- We implement our two protocols and perform a comprehensive comparison with the state-of-the-art JXT [22]. The experiment results demonstrate that both of our schemes outperform JXT in terms of query time and storage cost. Particularly, JXT+ (*resp.*, JXT++) brings a saving of 49% (*resp.*, 68%) in server-side storage cost and achieves a speedup of 51.7× (*resp.*, 54.3×) in search latency.

1.2 Technical Overview

Equi-Join Queries over Two Tables. Inspired by OXT [10], JXT [22] mainly construct two data structures, TSet and XSet, for each table in databases. Here, TSet serves as an inverted index for all attribute-value pairs, while XSet contains all combinations of record identifier and join-attribute value pair. The essential idea of JXT is to retrieve the records matching with specific attribute-value pairs from the TSet of table Tab_1 (*resp.* Tab_2), and then determine if the matching records from Tab_2 can be joined with those from Tab_1. This is achieved by checking whether the combinations of identifiers of ℓ_2 matching records from Tab_2 and join-attribute value pairs of ℓ_1 matching records from Tab_1 appear in the XSet. In addition, we note that the queried join attributes in JXT must be sent to the server to locate the corresponding entry in TSet, which leads to some extra leakages, i.e., the queried join attributes. Overall, JXT suffers from heavy join query complexity $O(\ell_1 \cdot \ell_2)$ and cannot handle equi-join queries because the join-attribute name and value are jointly stored into XSet for each record.

To support equi-join queries, a trivial solution is to decouple the join-attribute name from its corresponding value in JXT. Specifically, all join-attribute value pairs, in TSet and XSet, can be split into two components: join-attribute name $attr^*$ and join-attribute value w^*. Informally speaking, the original combination $(\mathsf{ind}, \langle attr^*, w^* \rangle)$ is replaced with a triple of $(\mathsf{ind}, attr^*, w^*)$, where the record identifier ind and join-attribute name $attr^*$ are from one table and join-attribute value w^* from another. Here, the only required change to achieve join with different join-attribute names is to include both queried join-attribute names in the search token. However, this approach still requires performing single-keyword search on both tables and then joining on their results, resulting in $O(\ell_1 \cdot \ell_2)$ query complexity. Meanwhile, the queried join attributes will be revealed.

We observe that equi-join can be achieved by checking the combination of join-attribute name and join-attribute value from different tables (e.g., $(attr_1^*, w_2^*)$). To further reduce join cost, our initial idea is to build TSet to store all combinations of attribute-value pair and join-attribute value (e.g., (w_1, w_1^*)) in each table, which is indexed by the combination of attribute-value pair and any possible join-attribute name. Then, the join can be done by checking the combination of $(w_2, attr_2^*, w_1^*)$ in XSet. Note that only a single table is queried, the query complexity is reduced to $O(\ell_1)$. Further, we introduce an additional data structure CSet indexed by the combination (e.g., $(w_1, attr_1^*, w_1^*)$) for each table, which stores "together" all the corresponding encrypted record identifiers. Thus, all the matching identifiers can be retrieved at a constant cost. In addition, our solution can hide the join-attribute information by binding it with non join-attribute value in TSet (i.e., $w||attr^*$).

Equi-Join Queries over Multiple Tables. An open question, "How to extend JXT to support join queries over *three or more* tables without join pre-computation?" is posed by Jutla et al. [22]. A straightforward answer is to divide the whole database into a collection of subsets including two designated tables and execute JXT protocol on them, then filter out the final result locally. Specifically, given a database with N tables $(\mathsf{Tab}_1, \ldots, \mathsf{Tab}_N)$, to achieve join queries over k tables $(k \leq N)$, the client performs JXT repeatedly on two-table pairs of $(\mathsf{Tab}_{t_1}, \mathsf{Tab}_{t_i})_{i \in [2,k]}$. After that, the client decrypts all the record identifiers matching each two-table join and determines the final result by obtaining the intersection. However, this naïve solution suffers from heavy query cost, i.e., $O(\sum_{i=2}^{k} \ell_1 \cdot \ell_i)$, where ℓ_i denotes the numbers of matching records in Tab_i.

To achieve efficient equi-join over multiple tables, our starting point is to extend our proposed JXT+ to support multiple-join queries. Specifically, the client first retrieves ℓ_1 entries matching w_1 from the TSet of Tab_1, and then checks whether the combinations $(w_2, attr_{t_2}^*, w_1^*), \ldots, (w_k, attr_{t_k}^*, w_1^*)$ include in XSet, where join-attribute value w_1^* is from Tab_1, attribute-value pairs $(w_2, \ldots, w_k)$ and attribute names $(attr_{t_2}^*, \ldots, attr_{t_k}^*)$ are from the rest $k-1$ tables. When all checks are passed, the client obtains the final result by retrieving all the matching record identifiers from all the k tables. Obviously, the query cost is only dependent on the matching record identifiers in the first table, i.e., $O((k-1) \cdot \ell_1)$. Nevertheless, this simple extension of JXT+ leaks non-trivial leakage. Particularly, the server might learn sub-query leakages[1], i.e., all the record identifiers matched with each two-table join. Additionally, it also leaks the frequency of join-attribute values associated with the queried attribute-value pairs, i.e., the number of occurrences of (w, w^*), which is more harmful to low-entropy join attributes. To address this issue, an intuitive solution is to pad all the occurrences of each (w, w^*) with dummy strings to the maximum volume $L_{max}[i][attr^*]$, which is the biggest occurrence of w^* in a table for all w. Thus, this naïve strategy suffers from a large server storage cost $2mnT + L_{max}mnT$.

To achieve better storage efficiency, our basic idea is to assign each pair $((w, attr^*, w^*), ct)$ in CSet to XOR filter [15, 37] and then pad the remaining empty locations with dummies. Thus, the total storage overhead is $3.23mnT$ at the worst case (cf. Section 6.1), where the storage of CSet is reduced from $L_{max}mnT$ to $1.23mnT$. Interestingly, sub-query results are obfuscated based on the padding strategy, preventing the server from accessing the exact records matching with any two-table join. Therefore, we can achieve equi-join queries over multiple tables without join pre-computation, while enjoying join over low-entropy attributes with $O(\ell_1)$ query complexity and $3.23mnT$ storage overhead.

1.3 Related Work

In 2000, Song et al. introduced the notion of symmetric searchable encryption (SSE), which enables the server to perform keyword-based search on encrypted data, while maintaining data and query

[1]When the query involves three or more tables, sub-query leakage (SRP) reveals whether some records are present in the join of two tables but not in the multi-table join. SRP is formalized as sub-query result pattern (SRP), as described in Section 5.

privacy. A variety of research progress has been done on enhanced security [5, 6, 8, 13, 26, 38], expressive queries [10, 11, 22, 24, 25, 32], and optimized performance [1, 9, 27, 36, 37]. Although SSE can achieve efficient search over encrypted documents, such as NoSQL databases and cloud storage, it is still challenging to design secure SQL queries over encrypted relational databases.

Hacigümüş et al. [18] firstly explored the study of SQL queries on encrypted relational databases, where each attribute domain is mapped into a series of non-overlapping data buckets and all the items of the corresponding bucket will be returned. Thus, it leads to high communication overhead and leaks the exact range of queried data. Popa et al. [32] presented a practical encrypted relational database system named CryptDB by integrating property-preserving encryption (PPE), such as deterministic encryption [2] and order-preserving encryption [4, 31]. That is, each column in the table will be encrypted with the composition of different types of PPE-encryption schemes (i.e., *onion* encryption). To perform certain SQL queries, the server decrypts the composition ciphertext by layer until the corresponding layer is reached. However, it has been reported in [16, 29] that PPE-based constructions leak non-trivial information, e.g., data sorting and frequency. Although existing leakage-abuse attacks [3, 17, 23, 30, 33] focus mainly on exact or range queries, Hoover et al. [20] take the first step to explore the SQL-oriented leakage-abuse attacks relying on access pattern, e.g., volume leakage from selections or joins. We note that the above attacks are mitigated effortlessly using the volume-hiding technique. Interestingly, our final protocol (JXT++) conceals the volumes of selection and join operations, which is not susceptible to the mentioned attacks. Nevertheless, it is essential to develop SQL queries over encrypted relational databases that reveal as little information as possible.

To design SQL queries with reduced leakage, Kamara et al. [25] presented a novel SQL queries scheme named SPX based on structure encryption, where all possible joins are pre-computed and stored in the encrypted database by heuristic normal form (HNF) representation. Although SPX reveals less leakage than PPE-style construction, it inevitably brings considerable storage blowup due to equi-join computation. Later, Cash et al. [11] introduced the notion of partially pre-computed joins and transferred certain join operations to the client. In addition, Hahn et al. [19] presented a fine-granularly secure join by adopting attribute-based encryption, that reveals only the equality pattern for records matching the selection criterion. Shafieinejad et al. [34] further designed secure equi-join for multiple queries from function-hiding inner product encryption, which leaks only the sum of the leakage of each query. Nevertheless, it suffers from performance bottlenecks due to expensive public-key operations.

Recently, Jutla et al. [22] extended the well-known OXT [10] to join queries over two tables, and presented a new join queries scheme dubbed JXT without join pre-computation. More concretely, the server performs single-keyword search for two queried tables separately and checks the existence of all combinations of the matching record identifiers and join-attribute value pairs across tables. However, JXT only supports natural join (i.e., the same name for join attributes) over two tables. Thus, it is imperative to design secure equi-join queries over multiple tables.

2 Preliminaries

In this section, we provide some required primitives throughout this paper, such as encrypted multi-map and join queries.

2.1 Symmetric Encryption

A symmetric encryption (SE) scheme is composed of three polynomial-time algorithms $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$:

$\mathsf{Gen}(1^\lambda)$: On input of security parameter λ, it outputs a secret key K.

$\mathsf{Enc}(K, m)$: On inputs of a secret key K and a message m, it outputs a ciphertext ct.

$\mathsf{Dec}(K, ct)$: On inputs a secret key K and a ciphertext ct, it outputs the corresponding message m or an error symbol $\perp$.

Correctness. A symmetric encryption scheme is computationally correct if for any message m and secret key K, the probability of the corresponding ciphertext ct can be correctly recovered is overwhelming, i.e., $\Pr[\mathsf{Dec}(K, ct) = m] = 1$.

Security. Informally, a standard IND-CPA secure SE scheme ensures that an adversary, with access to encryption oracle, cannot distinguish ciphertexts from two messages with the same length.

Definition 2.1. A symmetric encryption scheme $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is IND-CPA secure if for all PPT adversaries $\mathcal{A}$, its advantage

$$Adv^{\text{IND-CPA}}_{\mathsf{SE},\mathcal{A}}(\lambda) = |\Pr[\mathsf{Exp}^{\text{IND-CPA}}_{\mathsf{SE},\mathcal{A}}(\lambda) = 1] - 1/2|$$

is negligible in λ, where the experiment $\mathsf{Exp}^{\text{IND-CPA}}_{\mathsf{SE},\mathcal{A}}(\lambda)$ between a challenger and an adversary $\mathcal{A}$ is defined as follows.

Setup: The challenger generates a key $K \leftarrow \mathsf{Gen}(1^\lambda)$.

Query 1: The adversary $\mathcal{A}$ adaptively accesses the encryption oracle, meaning that when $\mathcal{A}$ queries on $m \in \mathcal{M}$, the challenger returns $ct \leftarrow \mathsf{Enc}(K, m)$.

Challenge: The adversary $\mathcal{A}$ sends two messages m_0 and m_1 with the same length to the challenger. Then the challenger generates two ciphertexts $c_0 \leftarrow \mathsf{Enc}(K, m_0)$ and $c_1 \leftarrow \mathsf{Enc}(K, m_1)$. After that, the challenger chooses a bit $b \in \{0, 1\}$ randomly and gives c_b to $\mathcal{A}$.

Query 2: The adversary $\mathcal{A}$ adaptively accesses the encryption oracle again, similar to Query 1, and subsequently it outputs a bit $b' \in \{0, 1\}$.

Guess: The adversary $\mathcal{A}$ outputs a bit $b' \in \{0, 1\}$. The experiment returns 1 if $b' = b$ and 0 otherwise.

2.2 XOR Filter

Graf and Lemire [15] first introduced the notion of XOR filter, which can be used to achieve membership checking with near-optimal storage overhead. Later, Wang et al. [37] provided its formal description and applied it to encrypted data search[2].

Let $\mathcal{U}$ be the universe of all possible inputs (i.e., strings) and B an array of λ-bit values. As shown in Algorithm 1, a (b, r)-XOR filter $\mathsf{XF} = (\mathsf{XF.Setup}, \mathsf{XF.Update}, \mathsf{XF.Query})$ consists of the following algorithms:

$\mathsf{XF.Setup}(b, r)$: It takes as input $b, r \in \mathbb{N}$, and samples a collection of universal hash functions $\mathcal{H} = \{h_t : \mathcal{U} \rightarrow [\frac{t}{r}b, \frac{t+1}{r}b)\}$, where

[2] As stated in [37], XOR filter can achieve data retrieving by replacing $H(x)$ with any data y at the corresponding positions $\{h_t(x)\}_{t \in [0, r-1]}$.

Algorithm 1 XOR Filter

```
XF.Setup(b, r)
1: B ← ∅, where |B| = b
2: H : U → {0,1}^λ
3: 𝓗={h_t : U → [t/r b, (t+1)/r b)}, where t ∈ [0, r − 1]
4: return (𝓗, B)

XF.Update(𝓗, B, S)
1: Stack ← Mapping Step(𝓗, S)
2: for (x, i) ∈ Stack do
3:     B[i] ← H(x)
4:     for t = 0 to r − 1 do
5:         if h_t(x) ≠ i then
6:             if B[h_t(x)] = null then
7:                 B[h_t(x)] ←$ {0,1}^λ
8:             end if
9:             B[i] ← B[i] ⊕ B[h_t(x)]
10:        end if
11:    end for
12: end for
13: for j = 0 to b − 1 do
14:    if B[j] = null then
15:        B[j] ←$ {0,1}^λ
16:    end if
17: end for
18: return B

XF.Query(𝓗, B, x)
1: R ← 0^λ
2: for t = 0 to r − 1 do
3:     R ← R ⊕ B[h_t(x)]
4: end for
5: return R

Mapping Step(𝓗, S)
1: Stack, Queue ← ∅
2: T ← ∅, where |T| = b
3: for x ∈ S do
4:     for t = 0 to r − 1 do
5:         T[h_t(x)] ← T[h_t(x)] ∪ {x}
6:     end for
7: end for
8: for i = 0 to b − 1 do
9:     if |T[i]| = 1 then
10:        Queue ← i
11:    end if
12: end for
13: while (Queue ≠ null) do
14:    i ← Queue
15:    x ← T[i]
16:    Stack ← (x, i)
17:    for t = 0 to r − 1 do
18:        T[h_t(x)] ← T[h_t(x)] \ {x}
19:        if |T[h_t(x)]| = 1 then
20:            Queue ← h_t(x)
21:        end if
22:    end for
23: end while
24: if |Stack| ≠ |S| then
25:    return False
26: end if
27: return True and Stack
```

$t \in [0, r-1]$. Finally, it outputs $\mathcal{H}$ and an initial empty array B of size b.

$\mathsf{XF.Update}(\mathcal{H}, B, S)$: It takes as input a family of hash functions $\mathcal{H}$, an empty array B, and a data set $S \subseteq \mathcal{U}$, then determines the order of inserting all elements by running Mapping Step, and pushes the elements to $Stack$ following the order. For each $(x, i) \in Stack$, it sets $B[i] \leftarrow x \bigoplus_{t \in [0, r-1] \setminus \{t'\}} B[h_t(x)]$, where $h_{t'}(x) = i$, and finally outputs array B.

$\mathsf{XF.Query}(\mathcal{H}, B, x)$: It takes as input $\mathcal{H}$, B as well as an element x, and returns $R = \bigoplus_{t=0}^{r-1} B[h_t(x)]$.

Perfect Completeness. An XOR filter is perfectly complete if for all integers $b, r \in \mathbb{N}$, all element set $S \subseteq \mathcal{U}$, $x \in S$ and $B_S \leftarrow \mathsf{XF.Update}(\mathcal{H}, B, S)$, it holds that $\Pr[\mathsf{XF.Query}(\mathcal{H}, B_S, x) = x] = 1$. This means that any inserted element can always be retrieved.

Parameter Choices. The storage cost of XOR filter is very close to the lower bound $O(n)$ while supporting efficient data query, where n denotes the size of set S. As indicated in [7, 28], the Mapping Step of XOR filter can assign all elements in set S to a set of $n = |S|$ edges generating an acyclic r-partite hypergraph, where the storage overhead is $C_r n + \beta$. The minimal value C_r is about 1.23 when $r = 3$. Thus, the size of B is $\lfloor 1.23n \rfloor + \beta$.

2.3 Encrypted Multi-Map

We recall multi-map, an abstract data structure for data retrieving is formalized in [14, 26]. Specifically, multi-map enables to store key/value pairs $\mathsf{MM} = \{(k, \vec{v}_k)\}$. Here, we denote by $\mathsf{MM}[k]$ all values associated with key k. A multi-map supports the following operations:

$\mathsf{Get}(k)$: On input of a key k, it outputs the associated tuple $\vec{v}_k = \mathsf{MM}[k]$.

$\mathsf{Put}(k, \vec{v}_k)$: On input of a key/value pair $(k, \vec{v}_k)$ and inserts it into multi-map, i.e., $\mathsf{MM}[k] = \vec{v}_k$.

To design encrypted search protocol, we further recall the notion of encrypted multi-map (EMM). Specifically, the syntax of response-hiding EMM [24] EMM = (Setup, Search) is presented as follows:

$\mathsf{Setup}(1^\lambda, \mathsf{MM})$: The client takes a security parameter λ and a multi-map MM as input, and outputs a secret key K and an encrypted multi-map EMM.

$\mathsf{Search}(\mathsf{K}, q; \mathsf{EMM})$: The client takes the secret key K as well as a query q as input and sends the corresponding search token tk_q to the server. Then the server performs search on EMM with tk_q and sends the encrypted search result $\mathsf{EMM}[q]$ to the client. Finally, the client recovers $\mathsf{MM}[q]$ from $\mathsf{EMM}[q]$ using the secret key K.

Generally, EMM can be classified into response-revealing and response-hiding. The former reveals the response in plaintext to the server, whereas the latter requires the client to decrypt the retrieved results locally. A response-revealing EMM scheme can be achieved by having the server retrieve the response in plaintext directly in Search algorithm.

Correctness. An encrypted multi-map scheme Σ is computationally correct if any adversary $\mathcal{A}$ has a negligible probability of winning the following game $\mathbf{Cor}^{\Sigma}_{\mathcal{A}}$. The adversary selects a multi-map MM and obtains EMM via $\mathsf{Setup}(1^{\lambda}, \mathsf{MM})$. The adversary then (non)-adaptively chooses a list of queries $\mathbf{q}$, and the game runs $\mathsf{Search}(\mathsf{K}, \mathbf{q}[i]; \mathsf{EMM})$ for all $i \in [|\mathbf{q}|]$ with client input $(\mathsf{K}, \mathbf{q}[i])$ and server input EMM. If any output from the client is not equal to the corresponding $\mathsf{MM}[\mathbf{q}[i]]$ ($i \in [|\mathbf{q}|]$), the game outputs 1; otherwise, it outputs 0. Σ is computationally correct if $\Pr[\mathbf{Cor}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$ for all $\mathcal{A}$.

Security. We define the non-adaptive security of an encrypted multi-map scheme using a leakage function $\mathcal{L}$, which represents the information revealed to an adversary during real-world scheme. Essentially, the scheme ensures that the adversary $\mathcal{A}$ cannot gain more information beyond what $\mathcal{L}$ reveals.

Definition 2.2 (Non-adaptive Security of Encrypted Multi-map). Let $\Sigma = (\mathsf{Setup}, \mathsf{Search})$ be an encrypted MM and $\mathcal{L}$ be its leakage function. We say Σ is $\mathcal{L}$-semantically secure against non-adaptive attacks if for all PPT adversaries $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ such that

$$\left|\Pr[\mathbf{Real}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]\right| \leq \mathsf{negl}(\lambda),$$

where $\mathbf{Real}^{\Sigma}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda)$ are defined as follows:

$\mathbf{Real}^{\Sigma}_{\mathcal{A}}(\lambda)$: $\mathcal{A}$ chooses a multi-map MM and a list of queries $\mathbf{q}$. The experiment runs $\mathsf{Setup}(1^{\lambda}, \mathsf{MM})$ and returns EMM to $\mathcal{A}$. For each $i \in [|\mathbf{q}|]$, the experiment runs $\mathsf{Search}(\mathsf{K}, \mathbf{q}[i]; \mathsf{EMM})$, and returns the transcript and client's output to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b as the output of this experiment.

$\mathbf{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda)$: $\mathcal{A}$ chooses a multi-map MM and a list of queries $\mathbf{q}$. The experiment runs $\mathcal{S}(\mathcal{L}(\mathsf{MM}, \mathbf{q}))$ and returns its outputs to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b as the output of this experiment.

To facilitate the description of our protocols, we further introduce TSet [10, 22], which essentially serves as a response-revealing EMM. It maintains a collection of fixed-size values indexed by keys, which can be accessed through tokens. Formally, a TSet instantiation is defined by three polynomial-time algorithms, described as follows:

$\mathsf{TSetSetup}(1^{\lambda}, \mathsf{T})$: It takes as input a security parameter λ and a multi-map $\mathsf{T} = \left\{\left(k, (v_k[1], \ldots, v_k[|\mathsf{T}[k]|])\right)\right\}_{k\in\mathbb{K}}$, where $\mathbb{K}$ is the set of keys in T, with each key $k \in \mathbb{K}$ having a corresponding list of values of equal bit length, i.e., $v_k[1], \ldots, v_k[|\mathsf{T}[k]|]$. It then outputs a secret key K_T and an EMM TSet.

$\mathsf{TSetGetTag}(K_T, k)$: It takes as input the secret key K_T and a queried key k, then outputs a search token stag.

$\mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$: It takes as input TSet and stag associated to k, and finally outputs $\mathsf{T}[k]$.

Note that TSet is a specialized type of EMM, and its correctness and security definitions are closely similar to those of EMM. Thus, we omit the details.

2.4 Join Queries over Encrypted Database

We recall the syntax of join queries over encrypted relational database presented in [22]. A relational database $\mathsf{DB} = \{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]}$ consists of a set of tables Tab_i, where W_i denotes all attribute-value pairs in Tab_i. For simplicity, assume that Tab_i has m_i rows (i.e., records) and n_i columns (i.e., attributes), Tab_i can be depicted as $\{(\mathsf{ind}_r, \{w_c\}_{c\in[n]})\}_{r\in[m]}$, where $\mathsf{ind}_r \in \{0,1\}^{\lambda}$ serves as the record identifier, and $w_c \in \{0,1\}^*$ denotes the attribute-value pair. Note that ind_r is used to retrieve the corresponding record in the outsourced encrypted database. We present some critical notations for join queries across tables as follows:

Record Identifiers: A record identifier, denoted as ind, is a unique value assigned to each record within a table Tab_i in a relational database. The identifier can be disclosed to the server that stores the relational database, enabling it to swiftly retrieve the corresponding encrypted record and send it to the client. We suppose that each record identifier in a table Tab_i is attached to the table number i throughout the paper. That is, there is no identical record identifier in two distinct tables.

Join Attributes: Consider a table Tab_i with a total of n attributes, among which T special attribute are designated as "join attributes" denoted by $\{attr^*_{i,t}\}_{t\in[T]}$. These join attributes with the size upper-bounded are chosen at setup and are used for join queries across tables.

Inverted Index: For each attribute-value pair $w \in \mathsf{W}_i$, $\mathsf{DB}_{\mathsf{Tab}_i}(w)$ refers to the set of identifiers of records matching w, specifically:

$$\mathsf{DB}_{\mathsf{Tab}_i}(w) = \{\mathsf{ind} \mid (\mathsf{ind}, w) \in \mathsf{Tab}_i\}.$$

The collection $\{\mathsf{DB}_{\mathsf{Tab}_i}(w)\}_{w\in\mathsf{W}_i}$ is named "inverted index" of the table Tab_i.

Inverted Join Index: For each attribute-value pair $w \in \mathsf{W}_i$, we denote $\mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(w)$ as the set of identifiers of records satisfying w, along with all the pairs of join-attribute/value in the same record. The formal description is as follows:

$$\mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(w) = \big\{(\mathsf{ind}, \{attr^*_t, w^*_t\}_{t\in[T]}) \mid (\mathsf{ind}, w) \in \mathsf{Tab}_i \wedge \forall t \in [T], (\mathsf{ind}, attr^*_t, w^*_t) \in \mathsf{Tab}_i\big\}.$$

All the $\{\mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(w)\}_{w\in\mathsf{W}_i}$ is called as "inverted join index" of table Tab_i.

Join Query: A join query over k tables $\mathsf{Tab}_{t_1}, \ldots, \mathsf{Tab}_{t_k}$ with corresponding attribute-value pair sets $\mathsf{W}_{t_1}, ..., \mathsf{W}_{t_k}$, respectively, is specified by a tuple

$$q = (\{t_1, ..., t_k\}, \{w_1, ..., w_k\}, \{attr^*_{t_1}, ..., attr^*_{t_k}\})$$

where $w_i \in \mathsf{W}_{t_i}$, and $attr^*_{t_i}$ is a join attribute of table Tab_{t_i} which defines the join relation across the tables for the query q.

We write $\mathsf{DB}(q)$ to be the collection of tuples of the form $(\mathbf{ind}_{t_1}, \ldots, \mathbf{ind}_{t_k})$ that satisfy the query q. Each tuple consists of k record identifier sets, each originating from $\mathsf{Tab}_{t_1}, \ldots, \mathsf{Tab}_{t_k}$, respectively. Formally, for each $(\mathbf{ind}_{t_1}, ..., \mathbf{ind}_{t_k}) \in \mathsf{DB}(q)$, this means that the following conditions hold simultaneously:

$$\bigwedge_{\substack{i\in[k]\\ j\in[|\mathbf{ind}_{t_i}|]}} (\mathbf{ind}_{t_i}[j], w_i) \in \mathsf{Tab}_{t_i} \text{ and } \exists\, \gamma \text{ s.t. } \bigwedge_{\substack{i\in[k]\\ j\in[|\mathbf{ind}_{t_i}|]}} (\mathbf{ind}_{t_i}[j], \langle attr^*_{t_i}, \gamma\rangle) \in \mathsf{Tab}_{t_i}$$

Algorithm 2 Equi-join Queries over Two Tables (JXT+)

EDBSetup$(1^\lambda, \mathsf{DB})$

1: $K_z, K_w, K_r, K_{\text{enc}} \xleftarrow{\$} \{0,1\}^\lambda$, $\mathsf{DB} = \{\mathsf{Tab}_i, \mathsf{W}_i\}_{i \in [N]}$
2: **for** $i = 1$ to N **do**
3: $\quad \mathsf{T}_i \leftarrow$ empty array, $\mathsf{XSet}[i] \leftarrow \emptyset$
4: $\quad \mathsf{CSet}[i] \leftarrow$ empty multi-map
5: $\quad$ **for** $w \in \mathsf{W}_i$ **do**
6: $\quad\quad \mathsf{cnt} \leftarrow 1$
7: $\quad\quad Z_0 \leftarrow F(K_z, w||0)$
8: $\quad\quad K_{\text{enc},w} \leftarrow F(K_{\text{enc}}, w)$
9: $\quad\quad$ **for** $(\mathsf{ind}, \{attr_t^*, w_t^*\}_{t \in [T]}) \in \mathsf{DB}_{\mathsf{Tab}_i}^{\mathsf{Join}}(w)$ **do**
10: $\quad\quad\quad Z_{\mathsf{cnt}} \leftarrow F(K_z, w||\mathsf{cnt})$
11: $\quad\quad\quad ct \leftarrow \mathsf{Enc}(K_{\text{enc},w}, \mathsf{ind})$
12: $\quad\quad\quad$ **for** $t \in [T]$ **do**
13: $\quad\quad\quad\quad y \leftarrow F(K_w, w_t^*) \oplus Z_{\mathsf{cnt}}$
14: $\quad\quad\quad\quad$ Append y to $\mathsf{T}_i[w||attr_t^*]$
15: $\quad\quad\quad\quad \mathsf{xtag} \leftarrow F(K_r, attr_t^*) \oplus F(K_w, w_t^*) \oplus Z_0$
16: $\quad\quad\quad\quad \mathsf{XSet}[i] \leftarrow \mathsf{XSet}[i] \cup \{\mathsf{xtag}\}$
17: $\quad\quad\quad\quad \mathsf{CSet}[i] \leftarrow \mathsf{CSet}[i].\mathsf{Put}(\mathsf{xtag}, ct)$
18: $\quad\quad\quad$ **end for**
19: $\quad\quad\quad \mathsf{cnt} \leftarrow \mathsf{cnt} + 1$
20: $\quad\quad$ **end for**
21: $\quad$ **end for**
22: **end for**
23: $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathsf{T}_1||...||\mathsf{T}_N)$
24: $\mathsf{K} \leftarrow (K_z, K_w, K_r, K_T, K_{\text{enc}})$, $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$
25: **return** $(\mathsf{K}; \mathsf{EDB})$

Search$(\mathsf{K}, q; \mathsf{EDB})$

Client:

1: $(K_z, K_r, K_T, K_{\text{enc}}) \leftarrow \mathsf{K}$, $(\{t_1, t_2\}, \{w_1, w_2\}, \{attr_{t_1}^*, attr_{t_2}^*\}) \leftarrow q$
2: $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, (t_1, w_1||attr_{t_1}^*))$
3: Send $(\{t_1, t_2\}, \mathsf{stag})$ to the server
4: **for** $\mathsf{cnt} = 1, 2 \ldots$ until server sends **stop do**
5: $\quad \mathsf{xjointoken}_1[\mathsf{cnt}] \leftarrow F(K_z, w_1||0) \oplus F(K_r, attr_{t_1}^*) \oplus F(K_z, w_1||\mathsf{cnt})$
6: $\quad \mathsf{xjointoken}_2[\mathsf{cnt}] \leftarrow F(K_z, w_2||0) \oplus F(K_r, attr_{t_2}^*) \oplus F(K_z, w_1||\mathsf{cnt})$
7: $\quad$ Send $\mathsf{xjointoken}_1[\mathsf{cnt}]$ and $\mathsf{xjointoken}_2[\mathsf{cnt}]$ to the server
8: **end for**

Server:

9: $(\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet}) \leftarrow \mathsf{EDB}$
10: $\mathsf{T}_{t_1}[w_1||attr_{t_1}^*] \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$
11: Parse $\mathsf{T}_{t_1}[w_1||attr_{t_1}^*] = (y_1, y_2, \ldots, y_{|\mathsf{T}_{t_1}[w_1||attr_{t_1}^*]|})$
12: **for** $\mathsf{cnt} = 1$ to $|\mathsf{T}_{t_1}[w_1||attr_{t_1}^*]|$ **do**
13: $\quad$ **if** $\mathsf{cnt} = |\mathsf{T}_{t_1}[w_1||attr_{t_1}^*]|$ **then**
14: $\quad\quad$ Send **stop** to the client
15: $\quad$ **end if**
16: $\quad \mathsf{xtoken}_2 \leftarrow \mathsf{xjointoken}_2[\mathsf{cnt}] \oplus y_{\mathsf{cnt}}$
17: $\quad$ **if** $\mathsf{xtoken}_2 \notin \mathsf{XSet}[t_2]$ **then**
18: $\quad\quad$ **break**
19: $\quad$ **end if**
20: $\quad \mathsf{xtoken}_1 \leftarrow \mathsf{xjointoken}_1[\mathsf{cnt}] \oplus y_{\mathsf{cnt}}$
21: $\quad \mathbf{ct}_1 \leftarrow \mathsf{CSet}[t_1].\mathsf{Get}(\mathsf{xtoken}_1)$
22: $\quad \mathbf{ct}_2 \leftarrow \mathsf{CSet}[t_2].\mathsf{Get}(\mathsf{xtoken}_2)$
23: $\quad \mathsf{CT} \leftarrow \mathsf{CT} \cup \{(\mathbf{ct}_1, \mathbf{ct}_2)\}$
24: **end for**
25: Send CT to the client

Client:

26: $K_{\text{enc},w_1} \leftarrow F(K_{\text{enc}}, w_1)$, $K_{\text{enc},w_2} \leftarrow F(K_{\text{enc}}, w_2)$, $\mathsf{Res} \leftarrow \emptyset$
27: **for** $(\mathbf{ct}_1, \mathbf{ct}_2) \in \mathsf{CT}$ **do**
28: $\quad$ **for** $i = 1$ to 2 **do**
29: $\quad\quad$ **for** $j = 1$ to $|\mathbf{ct}_i|$ **do**
30: $\quad\quad\quad \mathsf{Res}[i][j] \leftarrow \mathsf{Dec}(K_{\text{enc},w_i}, \mathbf{ct}_i[j])$
31: $\quad\quad$ **end for**
32: $\quad$ **end for**
33: **end for**
34: **return** Res

3 Equi-Join Queries over Two Tables

In this section, we present JXT+, a new equi-join query scheme over two tables without join pre-computation, which enjoys better query and storage efficiency than the state-of-the-art JXT [22].

Assume that $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is pseudorandom function, $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is IND-CPA secure symmetric encryption scheme, $\mathsf{TSet} = (\mathsf{TSetSetup}, \mathsf{TSetGetTag}, \mathsf{TSetRetrieve})$ is a non-interactive response-revealing EMM, and CSet is a multi-map. Our construction is comprised of one algorithm EDBSetup and one protocol Search in Algorithm 8 and the details of JXT+ are shown as follows:

$\mathsf{EDBSetup}(1^\lambda, \mathsf{DB})$: The client first randomly picks secret keys $K_z, K_w, K_r, K_{\text{enc}} \in \{0,1\}^\lambda$ for PRF F, and parses the database DB as $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i \in [N]}$, where W_i denotes the set of all attribute-value pairs in table Tab_i. Then the client initializes an empty array T_i, an empty set $\mathsf{XSet}[i]$ and an empty multi-map $\mathsf{CSet}[i]$ for each table Tab_i. After that, the client inserts each pair $(w, \mathsf{DB}_{\mathsf{Tab}_i}^{\mathsf{Join}}(w))$ of Tab_i into encrypted database EDB (cf. line 5-21). More concretely, for each pair of $(w, attr_t^*)$, the client first computes all the values $\{F(K_w, w_t^*) \oplus F(K_z, w||\mathsf{cnt})\}$ and stores them at location $\mathsf{T}_i[w||attr_t^*]$, which can be used to determine the number of record matching with w. Then, all $\{\mathsf{xtag}\}$ for each tuple $(w, attr_t^*, w_t^*)$ are inserted into $\mathsf{XSet}[i]$, which is the key component to achieve efficient euqi-join across two tables. Moreover, an additional data structure $\mathsf{CSet}[i]$ is used to store all pairs (xtag, ct). Finally, it outputs the secret key $\mathsf{K} = (K_z, K_w, K_r, K_T, K_{\text{enc}})$ and encrypted database $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$.

$\mathsf{Search}(K, q; \mathsf{EDB})$: To perform equi-join query q over Tab_{t_1} and Tab_{t_2}, the client sends stag for pair $(w_1, attr_{t_1}^*)$ [3] and table indices $\{t_1, t_2\}$, along with $\mathsf{xjointoken}_1$ and $\mathsf{xjointoken}_2$ arrays until instructed by the server to stop [4]. Next, the server retrieves the corresponding values $\mathsf{T}_{t_1}[w_1||attr_{t_1}^*]$ by calling TSetRetrieve

[3]It is unnecessary to reveal the join-attribute to the server in our protocol, while it will leak to the server for determining the desirable entry in JXT.

[4]JXT+ has only a single round of interaction similar as OXT and JXT, and its communication cost is composed of $(\{t_1, t_2\}, \mathsf{stag}, \{(\mathsf{xjointoken}_1[1], \mathsf{xjointoken}_2[1]), \ldots, (\mathsf{xjointoken}_1[\mathsf{cnt}], \mathsf{xjointoken}_2[\mathsf{cnt}])\})$.

Algorithm 3 Equi-join Queries over Multiple tables (JXT++)

EDBSetup$(1^\lambda, \mathrm{DB})$

1: $K_z, K_{z'}, K_r, K_w, K_c, K_{\mathrm{enc}} \leftarrow \{0,1\}^\lambda$, $\mathrm{DB} = \{\mathrm{Tab}_i, \mathrm{W}_i\}_{i \in [N]}$
2: **for** $i = 1$ to N **do**
3: $\quad \mathrm{T}_i \leftarrow$ empty array, $\mathrm{XSet}[i] \leftarrow \emptyset$, $\mathbf{M} \leftarrow$ empty map
4: $\quad$ **for** $w \in \mathrm{W}_i$ **do**
5: $\quad\quad \mathbf{C} \leftarrow$ empty map
6: $\quad\quad Z \leftarrow F(K_z, w)$, $Z' \leftarrow F(K_{z'}, w)$
7: $\quad\quad K_{\mathrm{enc},w} \leftarrow F(K_{\mathrm{enc}}, w)$
8: $\quad\quad$ **for** $(\mathrm{ind}, \{attr_t^*, w_t^*\}_{t \in [T]}) \in \mathrm{DB}^{\mathrm{Join}}_{\mathrm{Tab}_i}(w)$ **do**
9: $\quad\quad\quad ct \leftarrow \mathrm{Enc}(K_{\mathrm{enc},w}, \mathrm{ind})$
10: $\quad\quad\quad$ **for** $t \in [T]$ **do**
11: $\quad\quad\quad\quad$ **if** $\mathbf{C}[attr_t^* || w_t^*] = null$ **then**
12: $\quad\quad\quad\quad\quad \mathbf{C}[attr_t^* || w_t^*] \leftarrow [L_{max}[i][attr_t^*]]$
13: $\quad\quad\quad\quad\quad y \leftarrow F(K_w, w_t^*) \oplus Z$
14: $\quad\quad\quad\quad\quad$ Append y to $\mathrm{T}_i[w || attr_t^*]$
15: $\quad\quad\quad\quad\quad \mathrm{xtag} \leftarrow F(K_r, attr_t^*) \oplus F(K_w, w_t^*) \oplus Z' \oplus F(K_c, 1)$
16: $\quad\quad\quad\quad\quad \mathrm{XSet}[i] \leftarrow \mathrm{XSet}[i] \cup \{\mathrm{xtag}\}$
17: $\quad\quad\quad\quad$ **end if**
18: $\quad\quad\quad\quad$ Randomly choose $\mathrm{cnt} \in \mathbf{C}[attr_t^* || w_t^*]$
19: $\quad\quad\quad\quad \mathbf{C}[attr_t^* || w_t^*] \leftarrow \mathbf{C}[attr_t^* || w_t^*] \backslash \{\mathrm{cnt}\}$
20: $\quad\quad\quad\quad \mathrm{xtag} \leftarrow F(K_r, attr_t^*) \oplus F(K_w, w_t^*) \oplus Z' \oplus F(K_c, \mathrm{cnt})$
21: $\quad\quad\quad\quad \mathbf{M}[\mathrm{xtag}] \leftarrow ct$
22: $\quad\quad\quad$ **end for**
23: $\quad\quad$ **end for**
24: $\quad$ **end for**
25: $\quad (\mathcal{H}, \mathrm{CSet}[i]) \leftarrow \mathrm{XF.Setup}(1.23|\mathbf{M}| + \beta, r)$
26: $\quad \mathrm{CSet}[i] \leftarrow \mathrm{XF.Update}(\mathcal{H}, \mathbf{M}, \mathrm{CSet}[i])$
27: **end for**
28: $(\mathrm{TSet}, K_T) \leftarrow \mathrm{TSetSetup}(1^\lambda, \mathrm{T}_1 || ... || \mathrm{T}_N)$
29: $\mathrm{K} \leftarrow (K_z, K_{z'}, K_r, K_w, K_c, K_{\mathrm{enc}}, K_T)$, $\mathrm{EDB} \leftarrow (\mathrm{TSet}, \mathrm{XSet}, \mathrm{CSet})$
30: **return** $(\mathrm{K}; \mathrm{EDB})$

Search$(\mathrm{K}, q; \mathrm{EDB})$

Client:

1: $(K_z, K_{z'}, K_r, K_c, K_{\mathrm{enc}}, K_T) \leftarrow \mathrm{K}$
2: $(\{t_1, \ldots, t_k\}, \{w_1, \ldots, w_k\}, \{attr_{t_1}^*, \ldots, attr_{t_k}^*\}) \leftarrow q$
3: $\mathrm{stag} \leftarrow \mathrm{TSetGetTag}(K_T, (t_1, w_1 || attr_{t_1}^*))$
4: Send $(\{t_1, \ldots, t_k\}, \mathrm{stag})$ to the server
5: **for** $i = 1$ to k **do**
6: $\quad$ **for** $\mathrm{cnt} = 1$ to $L_{max}[i][attr_{t_i}^*]$ **do**
7: $\quad\quad \mathrm{xjointoken}[i][\mathrm{cnt}] \leftarrow F(K_{z'}, w_i) \oplus F(K_r, attr_{t_i}^*) \oplus F(K_z, w_1) \oplus F(K_c, \mathrm{cnt})$
8: $\quad$ **end for**
9: **end for**
10: Send xjointoken to the server

Server:

11: $(\mathrm{TSet}, \mathrm{XSet}, \mathrm{CSet}) \leftarrow \mathrm{EDB}$
12: $\mathrm{T}_{t_1}[w_1 || attr_{t_1}^*] \leftarrow \mathrm{TSetRetrieve}(\mathrm{TSet}, \mathrm{stag})$
13: Parse $\mathrm{T}_{t_1}[w_1 || attr_{t_1}^*] = (y_1, y_2, \ldots, y_{|\mathrm{T}_{t_1}[w_1 || attr_{t_1}^*]|})$
14: **for** $\mathrm{cnt} = 1$ to $|\mathrm{T}_{t_1}[w_1 || attr_{t_1}^*]|$ **do**
15: $\quad$ **for** $i = 2$ to k **do**
16: $\quad\quad \mathrm{xtoken}[i][1] \leftarrow \mathrm{xjointoken}[i][1] \oplus y_{\mathrm{cnt}}$
17: $\quad\quad$ **if** $\mathrm{xtoken}[i][1] \notin \mathrm{XSet}[t_i]$ **then**
18: $\quad\quad\quad$ **break**
19: $\quad\quad$ **end if**
20: $\quad\quad$ **if** $i = k$ **then**
21: $\quad\quad\quad$ **for** $j = 1$ to k **do**
22: $\quad\quad\quad\quad$ **for** $c = 1$ to $|\mathrm{xjointoken}[j]|$ **do**
23: $\quad\quad\quad\quad\quad \mathrm{xtoken}[j][c] \leftarrow \mathrm{xjointoken}[j][c] \oplus y_{\mathrm{cnt}}$
24: $\quad\quad\quad\quad\quad \mathbf{ct}_j[c] \leftarrow \mathrm{XF.Search}(\mathcal{H}, \mathrm{CSet}[t_j], \mathrm{xtoken}[j][c])$
25: $\quad\quad\quad\quad$ **end for**
26: $\quad\quad\quad$ **end for**
27: $\quad\quad\quad \mathrm{CT} \leftarrow \mathrm{CT} \cup \{(\mathbf{ct}_1, \ldots, \mathbf{ct}_k)\}$
28: $\quad\quad$ **end if**
29: $\quad$ **end for**
30: **end for**
31: Send CT to the client

Client:

32: $\mathrm{Res} \leftarrow \emptyset$
33: **for** $i = 1$ to k **do**
34: $\quad K_{\mathrm{enc},w_i} \leftarrow F(K_{\mathrm{enc}}, w_i)$
35: **end for**
36: **for** $(\mathbf{ct}_1, \ldots, \mathbf{ct}_k) \in \mathrm{CT}$ **do**
37: $\quad$ **for** $i = 1$ to k **do**
38: $\quad\quad$ **for** $j = 1$ to $|\mathbf{ct}_i|$ **do**
39: $\quad\quad\quad \mathrm{Res}[i][j] \leftarrow \mathrm{Dec}(K_{\mathrm{enc},w_i}, \mathbf{ct}_i[j])$
40: $\quad\quad$ **end for**
41: $\quad$ **end for**
42: **end for**
43: **Return** Res

and filters out the final result by checking whether the combination of $(w_2, attr_2^*, w_1^*,)$ is in XSet. Specifically, the server computes $\mathrm{xtoken}_2 = \mathrm{xjointoken}_2[i] \oplus y_i$ for each value in $\mathrm{T}_{t_1}[w_1 || attr_{t_1}^*]$ and tests whether $\mathrm{xtoken}_2 \in \mathrm{XSet}$. If hold, the server returns the matching encrypted identifiers CT associated with xtoken_1 and xtoken_2, where $\mathrm{xtoken}_1 = \mathrm{xjointoken}_1[i] \oplus y_i$. For each $(\mathbf{ct}_1, \mathbf{ct}_2) \in \mathrm{CT}$, the client recovers the matching record identifiers $\mathrm{Res} = \{(\mathbf{ind}_1, \mathbf{ind}_2) : \mathbf{ind}_1 = \mathrm{Dec}(K_{\mathrm{enc},w_1}, \mathbf{ct}_1), \mathrm{ind}_2 = \mathrm{Dec}(K_{\mathrm{enc},w_2}, \mathbf{ct}_2)\}$.

REMARK 1. *Here, we briefly present a variant of JXT+, named FJXT+, which enables to retrieve all rows matching* $attr_{t_1}^* = attr_{t_2}^*$ *without giving specific attribute-value pairs* w_1 *and* w_2*. Specifically, to achieve full equi-join without a filter, the only required change for FJXT+ is to insert an extra attribute column* $attr_0^i$ *for each table* Tab_i *filled with merely "#" at setup phase. When performing query* $q = (\{t_1, t_2\}, \{w_1 = attr_0^1 || \#, w_2 = attr_0^2 || \#\}, \{attr_{t_1}^*, attr_{t_2}^*\})$*, the client generates stag for pair* $(w_1, attr_{t_1}^*)$*, and the server can retrieve all rows (i.e., records) matching* $attr_{t_1}^*$ *from* TSet*, then the server can check if the retrieved* TSet *entries satisfy the join query (cf. line 12-24).*

To reflect the influence of change, we give a theoretical analysis of FJXT+ in terms of storage and query efficiency. For simplicity, assume that each table has m records and n attributes, with T join-attribute. Similar to JXT+, the total storage size of FJXT+ is $3m(n+1)T$ *in the worst case due to the insertion of an additional attribute column. For query overhead, the client generates a token for the pair of* $(w_1 = attr_0^1 || \#, attr_{t_1}^*)$ *and retrieves all the m rows (i.e., no filtering), then generates* $2m$ *cross-tokens* xjointoken*. Thus, the computation*

complexity for token generation is $O(m)Prf$. In addition, to retrieve all rows (m) in Tab_{t_1}, the server needs to perform $O(m)$ Prf and $O(m)$ XOR operations for join with $attr^*_{t_2}$. Then the server and the client are required to compute $O(|R|)$ hash and $O(|R|)$ decryption operations, respectively. Overall, the query overhead is $O(m)Prf + O(|R|)Dec$ for the client and $O(m)(Prf + Xor) + O(|R|)H$ for the server.

4 Equi-Join Queries over Multiple Tables

In this section, we propose JXT++, the first equi-join query scheme over multiple tables without join pre-computation, by extending our basic scheme JXT+, which can support join of arbitrary low-entropy attributes and mitigate the sub-query leakage.

Let $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ be a PRF, $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ an IND-CPA secure SE scheme, and $\Sigma = (\mathsf{TSetSetup}, \mathsf{TSetGetTag}, \mathsf{TSetRetrieve})$ a non-interactive response-revealing EMM. Given any database $\mathsf{DB} = \{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]}$, the client initializes a two-dimensional array L_{max} with size $N \times T$ to store the maximum occurrence number of each pair $(w, w^*_t)_{w\in\mathsf{W}_i, t\in[T]}$, where w^*_t is the corresponding value of join attribute $attr^*_t$ in table Tab_i. JXT++ consists of one algorithm EDBSetup and one protocol Search in Algorithm 3 and the details are described as follows:

$\mathsf{EDBSetup}(1^\lambda, \mathsf{DB})$: The client first parses the database DB as $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]}$, and randomly picks keys $K_z, K_{z'}, K_r, K_w, K_c, K_{\mathrm{enc}} \in \{0,1\}^\lambda$ for PRF F. Then it initializes T_i as an empty array, $\mathsf{XSet}[i]$ as an empty set, and $\mathbf{M}$ as an empty map for each table Tab_i. Later, the client inserts all the pairs $(w, \mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(w))$ of Tab_i into EDB (cf. line 4-26). Specifically, the client computes all values $\{F(K_z, w^*_t) \oplus F(K_z, w)\}$ and appends into $\mathsf{T}_i[w||attr^*_t]$ for each pair of $(w, attr^*_t)$. The client further computes $\mathrm{xtag} = F(K_r, attr^*_t) \oplus F(K_w, w^*_t) \oplus Z' \oplus F(K_c, 1)$ for $(w, w^*_t, attr^*_t)$ and stores all $\{\mathrm{xtag}\}$ into $\mathsf{XSet}[i]$. In addition, each encrypted identifier ct is stored into map $\mathbf{M}$ indexing by a unique token (i.e., $\mathrm{xtag} = F(K_r, attr^*_t) \oplus F(K_w, w^*_t) \oplus Z' \oplus F(K_c, \mathrm{cnt})$). After that, all pairs (xtag, ct) in $\mathbf{M}$ is used to generate array $\mathsf{CSet}[i]$ by invoking XF.Update, which can hide the frequency of pair (w, w^*_t). Finally, it outputs the secret key $\mathsf{K} = (K_z, K_{z'}, K_r, K_w, K_c, K_{\mathrm{enc}})$ and the encrypted database $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$.

$\mathsf{Search}(K, q; \mathsf{EDB})$: To perform equi-join over k tables, the client sends stag for pair $(w_1, attr^*_{t_1})$, queried table indices $\{t_1, \ldots, t_k\}$, and xjointoken array to the server. Then, the server first retrieves all values $\mathsf{T}_{t_1}[w_1||attr^*_{t_1}]$ by invoking TSetRetrieve. Next, the server generates $k-1$ tokens for each retrieved value y_{cnt} (i.e., $\{\mathsf{xtoken}[i][1] = \mathsf{xjointoken}[i][1] \oplus y_{\mathrm{cnt}}\}_{i\in[2,k]}$), which is used to determine whether it is in the join of Tab_{t_1} and Tab_{t_i}. After all checkings are successful, it computes $\mathsf{xtoken}[j][c] = \mathsf{xjointoken}[j][c] \oplus y_{\mathrm{cnt}}$ and retrieves the corresponding ciphertext of record identifier $\mathbf{ct}_j[c]$ by invoking XF.Search (cf. line 20-28). It then returns all ciphertexts $\{(\mathbf{ct}_1, \ldots, \mathbf{ct}_k)\}$ to the client. Finally, the client decrypts them and obtains the plaintexts $\mathsf{Res} = \{(\mathbf{ind}_1, \ldots, \mathbf{ind}_k) : \mathbf{ind}_1 = \mathsf{Dec}(K_{\mathrm{enc},w_1}, \mathbf{ct}_1), \ldots, \mathbf{ind}_k = \mathsf{Dec}(K_{\mathrm{enc},w_k}, \mathbf{ct}_k)\}$.

5 Security Analysis

Assume that a database $\mathsf{DB} = \{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]}$ consists of N tables, and each Tab_i has m_i rows and n_i columns. We define a sequence of Q queries as $\mathbf{q}$, where the i-th query, for $1 \le i \le Q$, is represented as $\mathbf{q}[i] = (\{\mathbf{t}_1[i], \ldots, \mathbf{t}_k[i]\}, \{\mathbf{w}_1[i], \ldots, \mathbf{w}_k[i]\}, \{\mathbf{attr}^*_{\mathbf{t}_1}[i], \ldots, \mathbf{attr}^*_{\mathbf{t}_k}[i]\})$. Then the leakage function $\mathcal{L}_{\mathrm{JXT+}} = (\mathsf{n}, \mathsf{RP}, \mathsf{EP}, \mathsf{SP}_1, \mathsf{JD})$ for JXT+ and $\mathcal{L}_{\mathrm{JXT++}} = (\mathsf{n}, \mathsf{RP}, \mathsf{EP}, \mathsf{JD})$ for JXT++ are defined as follows:

- n is the *size pattern of tables*, representing the size of each table in DB. Formally, $\mathsf{n}[i] = m_i n_i$ for $i \in [N]$.
- RP is the *result pattern*, which is the set of records matching each query. Formally, $\mathsf{RP}[i] = \mathsf{DB}(\mathbf{q}[i])$, $i \in [Q]$.
- $\mathsf{EP} = (\mathsf{EP}_1, \ldots, \mathsf{EP}_k)$ represents the *equality patterns* over $\mathbf{w}_1||\mathbf{attr}^*_{\mathbf{t}_1}, \ldots, \mathbf{w}_k||\mathbf{attr}^*_{\mathbf{t}_k}$, respectively. Specifically, EP_1 indicates which queries have equal combination of $\mathbf{w}_1[i]||\mathbf{attr}^*_{\mathbf{t}_1}[i]$ for $i \in [Q]$. Formally, we represent EP_1 as an integer vector and each integer refers to a unique combination. That is, $\mathsf{EP}_1[i] = \mathsf{EP}_1[j]$ if $(\mathbf{w}_1[i]||\mathbf{attr}^*_{\mathbf{t}_1}[i]) = (\mathbf{w}_1[j]||\mathbf{attr}^*_{\mathbf{t}_1}[j])$ for $i, j \in [Q]$ and $i \neq j$. $\mathsf{EP}_2, \ldots, \mathsf{EP}_k$ are defined similarly.
- $\mathsf{SP} = (\mathsf{SP}_1, \ldots, \mathsf{SP}_k)$ is the *size patterns* over the combinations $\mathbf{w}_1||\mathbf{attr}^*_{\mathbf{t}_1}, \ldots, \mathbf{w}_k||\mathbf{attr}^*_{\mathbf{t}_k}$. Concretely, SP_1 indicates the number of records matching $\mathbf{w}_1||\mathbf{attr}^*_{\mathbf{t}_1}$ in TSet during each join query. Formally, $\mathsf{SP}_1[i] = |\mathsf{T}_{\mathbf{t}_1[i]}[\mathbf{w}_1[i]||\mathbf{attr}^*_{\mathbf{t}_1}[i]]|$ for $i \in [Q]$.
- JD is the *join-attribute distribution pattern*, which mainly leaks the join-attribute values corresponding to the combination of attribute-value pair $\mathbf{w}_1[i]$ and join attribute $\mathbf{attr}^*_{\mathbf{t}_1}[i]$ in table $\mathsf{Tab}_{\mathbf{t}_1[i]}$. More formally, for each $i \in [Q]$, we denote $\mathsf{JD}[i]$ as a multi-set

$$\mathsf{JD}[i] = \{\mathrm{encode}(\mathrm{val}^*) : (\mathrm{ind}, \mathbf{w}_1[i]) \in \mathsf{Tab}_{\mathbf{t}_1[i]} \wedge (\mathrm{ind}, \langle \mathbf{attr}^*_{\mathbf{t}_1}[i], \mathrm{val}^* \rangle) \in \mathsf{Tab}_{\mathbf{t}_1[i]}\}.$$

Before presenting the formal security analysis of our schemes, we first recall the conditional intersection pattern (IP) leakage in [22] considering query on two tables, and introduce the sub-query result pattern (SRP) leakage.

- IP is the *conditional intersection pattern* which is a $Q \times Q$ table. Specifically, for each $i, j \in [Q]$, $\mathsf{IP}[i, j]$ is empty if one of the following conditions holds:
  - $(\mathbf{t}_1[i], \mathbf{t}_2[i], \mathbf{attr}^*_{\mathbf{t}_1}[i], \mathbf{attr}^*_{\mathbf{t}_2}[i]) \neq (\mathbf{t}_1[j], \mathbf{t}_2[j], \mathbf{attr}^*_{\mathbf{t}_1}[j], \mathbf{attr}^*_{\mathbf{t}_2}[j])$.
  - $\mathsf{JD}[i] \cap \mathsf{JD}[j] = \emptyset$.

  Otherwise, $\mathsf{IP}[i, j]$ is defined as the intersection of all record identifiers matching the attribute-value pair $\mathbf{w}_2[i]$ and $\mathbf{w}_2[j]$ in table $\mathsf{Tab}_{\mathbf{t}_2[i]}$. More formally, we have

$$\mathsf{IP}[i, j] = \mathsf{DB}_{\mathsf{Tab}_{\mathbf{t}_2[i]}}(\mathbf{w}_2[i]) \cap \mathsf{DB}_{\mathsf{Tab}_{\mathbf{t}_2[j]}}(\mathbf{w}_2[j]).$$

- SRP is the *sub-query result pattern*, which is represented as a collection of $Q \times (k-1)$ sets. Formally, for $i \in [Q]$, we first divide $\mathbf{q}[i]$ into $(k-1)$ sub-queries, that is, for $j \in [k-1]$,

$$\mathbf{q}_j[i] = (\{\mathbf{t}_1[i], \mathbf{t}_{j+1}[i]\}, \{\mathbf{w}_1[i], \mathbf{w}_{j+1}[i]\}, \{\mathbf{attr}^*_{\mathbf{t}_1}[i], \mathbf{attr}^*_{\mathbf{t}_{j+1}}[i]\}).$$

  Then $\mathsf{SRP}[i][j]$ is the search result for the j-th sub-query $\mathbf{q}_j[i]$. More formally, for $i \in [Q]$ and $j \in [k-1]$, we have $\mathsf{SRP}[i][j] = \mathsf{DB}(\mathbf{q}_j[i])$.

Leakage Comparison on Equi-join without Pre-computation: Similar to JXT, our proposed schemes JXT+ and JXT++ can support join queries without join pre-computation. So, we provide a detailed

leakage comparison of the three schemes, as shown in Table 2. Note that JXT and JXT+ support join query on two tables, while JXT++ works on multiple tables. In the following, we give the details.

All three schemes have the leakage of table size pattern n, which comes from the size of TSet. The equality pattern EP can be derived from the repetition of stag and xjointoken when performing a query for the same attribute-value pair w and $attr^*$. Specifically, in JXT, the client generates $\mathsf{stag}^{(1)}$ and $\mathsf{stag}^{(2)}$ for the queried attribute-value pair w_1 of Tab_{t_1} and w_2 of Tab_{t_2}, respectively. Thus JXT leaks the exact equality patterns of $\mathbf{w}_1$ and $\mathbf{w}_2$, being also denoted as EP_1 and EP_2. Additionally, the server in JXT proceeds to retrieve the search results for w_1 and w_2 based on $\mathsf{stag}^{(1)}$ and $\mathsf{stag}^{(2)}$, obviously leaking the size patterns SP_1 and SP_2 of $\mathbf{w}_1$ and $\mathbf{w}_2$, respectively. In contrast, JXT+ and JXT++ leak EP_1 and EP_2 but not reveal all the size pattern SP. Note that in JXT+, the server can retrieve all records matching $w_1||attr^*_{t_1}$, whereas in JXT++, the server only retrieves one copy of records with the same $(w_1, \langle attr^*_{t_1}, w^*_{t_1}\rangle)$. Therefore, JXT+ has SP_1 leakage, while JXT++ does not.

In the following, we first analyze the subtle leakage JD. For some query $\mathbf{q}[i]$ $(i \in [Q])$, $\mathsf{JD}[i]$ reflects the frequency distribution of join-attribute values w^* corresponding to the join attribute $\mathbf{attr}^*_{t_1}[i]$ in the records matching the attribute-value pair $\mathbf{w}_1[i]$ in $\mathsf{Tab}_{\mathbf{t}_1[i]}$. In JXT and JXT+, this leakage comes from the fact that the records share the same w^* in the i-th query (or the records in the i-th and j-th queries). Thus, for two different queries $\mathbf{q}[i]$ and $\mathbf{q}[j]$, the server may get the same xtoken corresponding to $(\mathbf{w}_2[i], w^*)$ to checking whether it is in the XSet. This situation may occur if a condition $(\mathbf{w}_2[i]||\mathbf{attr}^*_{\mathbf{t}_2}[i]) = (\mathbf{w}_2[j]||\mathbf{attr}^*_{\mathbf{t}_2}[j])$ holds. JXT++ leaks the JD in different queries for reasons similar to those in JXT and JXT+. However, JXT++ does not leak JD when considering a single query (e.g. the i-th query), since JXT++ stores only one copy of records with the same $(w_1, \langle attr^*_{t_1}, w^*_{t_1}\rangle)$.

JXT+ and JXT++ eliminate the IP leakage that is present in JXT. Recall that IP leakage reveals the intersection of records matching the attribute-value pair w_2 in the Tab_{t_2} in different queries with the condition that there exists two records from different queries matching w_1 have identical w^*. In JXT, this leakage comes from the fact that xtag is the encrypted version of the pair (ind, w^*). That is, given a specific w^*, the server can get the identical xtoken if two queries share the same ind. For clarity, we take an example to illustrate. For the i-th and j-th query, assume that there is one record matching $\mathbf{w}_1[i]$ and another record matching $\mathbf{w}_1[j]$ have a common w^*. In this case, if the server learns the xtoken for ind_i from the results of $\mathbf{w}_2[i]$ and ind_j from the results of $\mathbf{w}_2[j]$ are identical, then $\mathsf{ind}_i = \mathsf{ind}_j$. In contrast, JXT+ and JXT++ remove the IP leakage since the xtag is the encrypted version of the pair $(w, attr^*, w^*)$. Therefore, the server cannot get the identical value even when a record ind matches both $\mathbf{w}_2[i]$ and $\mathbf{w}_2[j]$.

SRP is a straightforward leakage for multi-table queries. Specifically, the idea of performing multi-table queries is that the server first retrieves the results from Tab_{t_1}, and then filters the results that satisfy the sub-queries for $\mathsf{Tab}_{t_2}, \cdots, \mathsf{Tab}_{t_k}$ individually. Consequently, it naturally reveals the results matching Tab_{t_1} and Tab_{t_i} for $2 \le i \le k$. However, we note that the search results in JXT++ contain some dummy values for each sub-query, thus it does not have the SRP leakage.

Table 2: Leakage comparison with JXT.

Scheme	n	RP	EP_1	EP_2	SP_1	SP_2	JD	IP	SRP
JXT [22]	●	●	●	●	●	●	●	●	–
JXT+	●	●	●	●	●	○	●	○	–
JXT++	●	○	●	●	○	○	◐	○	○

The symbols ●, ◐, and ○ denote fully, partially, and without revealing the leakage.

Theorem 1. *Our JXT+ protocol is $\mathcal{L}$-semantically-secure against non-adaptive attacks where the leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{JXT+}}(\mathsf{DB}, \mathbf{q}), \mathcal{L}_{\mathsf{T}}(\mathsf{DB}, \mathbf{t}_1, \mathbf{w}_1, \mathbf{attr}^*_{\mathbf{t}_1}))$, assuming that F is a secure PRF,* SE *is a standard IND-CPA secure symmetric encryption scheme, and* TSet *is a non-adaptively $\mathcal{L}_{\mathsf{T}}$-secure instantiation.*

Theorem 2. *Our JXT++ protocol is $\mathcal{L}$-semantically-secure against non-adaptive attacks where the leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{JXT++}}(\mathsf{DB}, \mathbf{q}), \mathcal{L}_{\mathsf{T}}(\mathsf{DB}, \mathbf{t}_1, \mathbf{w}_1, \mathbf{attr}^*_{\mathbf{t}_1}))$, assuming that F is a secure PRF,* SE *is a standard IND-CPA secure symmetric encryption scheme, and* TSet *is a non-adaptively $\mathcal{L}_{\mathsf{T}}$-secure instantiation.*

The proof of Theorem 1 is provided in Appendix A. Additionally, the proof of Theorem 2 is similar to that of Theorem 1 and is omitted. We remark that both of our protocols are also secure against adaptive attacks. Here, the adaptive security proof is essentially an extension of the non-adaptive proof. In brief, the key difference is that the adaptive security proof involves using the adaptive TSet simulator, XSet and CSet are be simulated to adaptively respond to the adversary's queries. Note that the proofs of adaptive security for our protocols are also similar to those of [22], so we omit the details.

6 Performance Evaluation

In this section, we provide a detailed asymptotic comparison between our proposed schemes and the most related works. We then describe the implementation configuration and present the detailed experimental results.

6.1 Theoretical Comparison

In the following, we present a performance comparison of JXT+ and JXT++ with SPX [25], CNR [11] and JXT [22] in terms of query and storage efficiency. For simplicity, we consider a database including two tables, $\mathsf{Tab}_{t_1}, \mathsf{Tab}_{t_2}$, each with m rows, n columns, and T join attributes, and suppose the query q = (Select $inds$ From $\mathsf{Tab}_{t_1}, \mathsf{Tab}_{t_2}$ Join On $attr^*_{t_1} = attr^*_{t_2}$ Where $w_1 \wedge w_2$). A summary of the comparison is given in Table 1.

Storage Overhead. We first focus on the storage size of SPX, CNR, JXT, JXT+, and JXT++. Note that the storage of JXT and our schemes is considered table-wise. Specifically, SPX produces three encrypted multi-maps EMM_R, EMM_C, EMM_V and an encrypted dictionary EDX. EMM_R is a row-wise representation of the database that maps the row's identifier to all the encrypted entries in that row. Similarly, EMM_C is a column-wise representation of the database that maps the column's identifier to the encrypted entries in that column. EMM_V maps each value in a column to the rows' identifiers containing that value. In other words, each of the three encrypted multi-maps encrypts every cell in a table, requiring $O(mn)$ storage. EDX consists of a set of multi-maps, and each includes the pre-computation of all the join results for a join attribute. Thus, the size of EDX depends on the number of join attributes and the

number of rows satisfying the join. In the best case, no results are satisfying the join, then EDX will be empty; while in the worst case, each join-attribute shares the common value in two tables, then the size of EDX is $O(m^2T)$. In summary, the storage size of SPX is $O(mn) + O(m^2T)$.

CNR generates an encrypted multi-map EM and a filter HS of a set SET. EM is similar to EMM_V in SPX that maps each value in every column to the row's identifiers containing that value, thus it requires $O(mn)$ storage. In contrast to EDX in SPX, SET in CNR contains the pre-computation of joining rows from each table separately to reduce the storage size. When the join attributes share a common value (i.e., worst case), then the size of SET is $O(mT)$. Hence, the overall storage size of CNR is $O(mn) + O(mT)$.

JXT generates an encrypted multi-map TSet and a set XSet. Specifically, for each attribute-value pair w in the database, TSet stores the encrypted record identifiers, the T combinations of record identifier and join attribute, as well as the T combinations of join-attribute value and attribute-value pair, denoted as $(ct, \{y_t, y'_t\}_{t\in[T]})$. This results in a storage overhead of $(2T + 1)mn$. XSet stores all the combinations of record identifiers and corresponding join-attribute value pairs in the database, having mT storage size. Therefore, the total storage size of JXT is $(2T + 1)mn + mT$.

JXT+ produces an encrypted multi-maps TSet, a set XSet and a multi-map CSet. For every attribute-value pair w and join attribute $attr^*$, TSet encrypts the pair of the attribute-value pair and corresponding join-attribute value for the join-attribute name (e.g., (w, w^*)). Hence the storage size of TSet is mnT. For each attribute-value pair w, XSet stores combinations of attribute-value pair, join-attribute name, and join-attribute value (e.g., $(w, attr^*, w^*)$). Note that only one copy is stored for the duplicated $(w, attr^*, w^*)$ in XSet. Specifically, it has a storage size of $\sum_{i\in[n],j\in[T]} m^-_{i,j}$, where $m^-_{i,j}$ denotes the number of distinct $(w, attr^*, w^*)$ for the i-th attribute column and the j-th join-attribute column. The corresponding sizes are mnT and nT for the worst and best cases, respectively. CSet is used to store all encrypted record identifiers indexed by $(w, attr^*, w^*)$ and has a storage size of mnT. Overall the total storage sizes of JXT+ are $3mnT$ and $2mnT$+nT for the worst and best cases, respectively.

Similar to JXT+, JXT++ produces an encrypted multi-map TSet, a set XSet and a map CSet. To hide the frequency of join-attribute values, both TSet and XSet in JXT++ will save a single copy for the duplicated $(w, attr^*, w^*)$. Besides, the CSet with the size of mnT is mapped to an XOR filter by padding dummy values for removing SRP leakage. This results in $1.23mnT$ storage overhead. Therefore, the total storage sizes of JXT++ are $3.23mnT$ and $1.23mnT$+$2nT$ for worst and best cases, respectively.

Search Efficiency. Next, we proceed to examine the efficiency of performing a query q = (Select *inds* From $\mathsf{Tab}_{t_1}, \mathsf{Tab}_{t_2}$ Join On $attr^*_{t_1} = attr^*_{t_2}$ Where $w_1 \wedge w_2$). During the search phase, the computational costs are divided between the client and the server. In detail, the client generates the search token and then the server performs a search to retrieve the search results and finally the client decrypts them.

In SPX, the client generates tokens for w_1, w_2 and $attr^*_{t_1} = attr^*_{t_2}$ respectively, resulting in $O(1)Prf$. Then the server queries EMM_V based on the tokens of w_1 and w_2 to recover ℓ_1 and ℓ_2 records; and queries EDX based on the tokens of $attr^*_{t_1} = attr^*_{t_2}$ to recover $|R_J|$ records, where $|R_J|$ is the number of records satisfying the join $attr^*_{t_1} = attr^*_{t_2}$; and performs intersection of the results to obtain the final encrypted results R. The decryption of R is performed on the client side. Thus, the query computation cost is $O(\ell_1 + \ell_2 + |R_J|)\mathsf{EMM}_{\mathsf{qry}}$ (server side) and $O(1)Prf + O(|R|)Dec$ (client side), respectively.

To reduce storage size, CNR introduces the technique of partially pre-computed joins at the cost of additional client-side computation. That is, the client requires constant PRF to generate the search token of $(w_1, w_2, attr^*_{t_1}, attr^*_{t_2})$. On receiving the search token, the server first retrieves the matching records (ℓ_1 and ℓ_2) from encrypted multi-map EM with $O(\ell_1+\ell_2)\mathsf{EMM}_{\mathsf{qry}}$ computation cost, and then checks whether each result of w_1 (*resp.* w_2) joins with some value of $attr^*_{t_2}$ (*resp.* $attr^*_{t_1}$) based on the filter HS by $O(\ell_1 + \ell_2)$ Prf operation. Therefore, the server computational cost is $O(\ell_1 + \ell_2)(\mathsf{EMM}_{\mathsf{qry}} + Prf)$ in total. Note that the client needs to decrypt the search result and perform join locally, so the search cost is $O(\ell_1 + \ell_2)Dec + O(\ell_1\ell_2)Join$.

Both of the mentioned schemes involve (full or partial) join pre-computation and lead to either high storage size or heavy client computational cost. In contrast, JXT, JXT+, and JXT++ construct table-wise index structures with joinable attributes, avoiding prohibitive join pre-computation. More concretely, the client in JXT generates search tokens for w_1 and w_2 and computes $\ell_1 + \ell_2$ cross-tokens xjointoken, incurring a cost of $O(\ell_1 + \ell_2)Prf$. On the server side, it retrieves ℓ_1 and ℓ_2 encrypted identifiers matching with w_1 and w_2, respectively, and then performs join with all possible combinations. Thus, the total search cost is $O(\ell_1 + \ell_2)\mathsf{EMM}_{\mathsf{qry}} + O(\ell_1\ell_2)Xor$. The decryption of identifiers is performed by the client and the entire cost is $O(\ell_1 + \ell_2)Prf + O(|R|)Dec$.

To further reduce join cost, JXT+ stores all combinations of attribute-value pair and join-attribute value (e.g., (w_1, w^*_1)) in each table and achieves the join by checking the combination of (w_2, $attr^*_{t_2}$, w^*_1) in XSet. Specifically, it spends $O(\ell_1)\mathsf{EMM}_{\mathsf{qry}}$ to retrieve the records matching $w_1||attr^*_{t_1}$, an additional $O(\ell_1)Xor$ to perform join with $(w_2, attr^*_{t_2})$, and $O(|R|)H$ to retrieve the matched encrypted record identifiers ct. The total query cost for the server is $O(\ell_1)(\mathsf{EMM}_{\mathsf{qry}} + Xor) + O(|R|)H$. In terms of client, the cost consists of token generation for pair $(w_1, attr^*_{t_1})$ and decryption on R search result. i.e., $O(\ell_1)Prf + O(|R|)Dec$.

JXT++ is derived from JXT+ to handle multiple tables. To hide the real number of occurrences of (w, w^*), all the occurrences of each pair of (w, w^*) are to pad with dummy strings to the maximum volume l_{max}. This results in $O(\ell_1)\mathsf{EMM}_{\mathsf{qry}}$+$O(\ell_1 l_{max})(Xor$+$H)$ in the worst case (Assuming that all l_1 records satisfying the query q) cost due to the retrieving of dummy data. While the client needs to generate tokens for pair $(w_1, attr^*_{t_1})$, compute cross-tokens xjointoken, and decrypt all the result. Its computation cost is $O(l_{max})Prf + O(\ell_1 l_{max})Dec$.

6.2 Implementation Configuration

We implement our schemes and JXT [22] by using JAVA, employing the JDK library for cryptographic operations like AES and SHA-256. In our experiments, we adopt such a database consisting of 6 tables, and each table owns 65,535 rows and 11 attribute columns (including

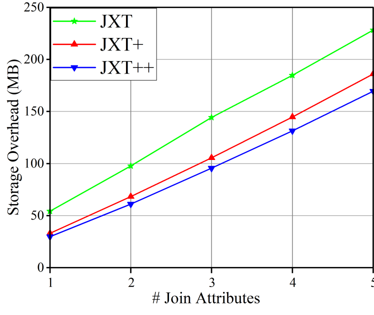

Figure 1: Storage Overhead Comparison

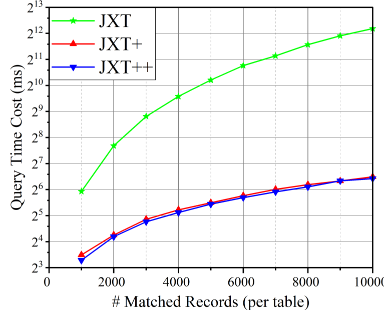

Figure 2: Query Time (all matched)

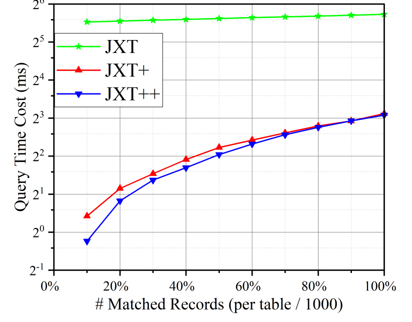

Figure 3: Query Time (partly matched)

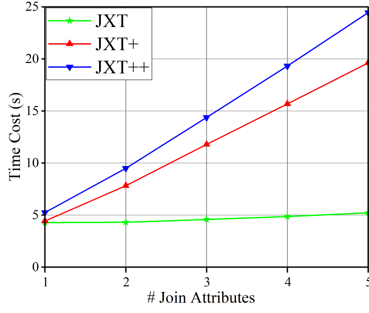

Figure 4: Setup Time Comparison

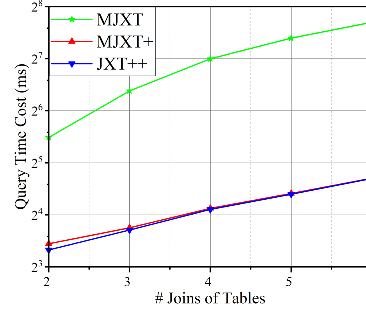

Figure 5: Query Time With Tables

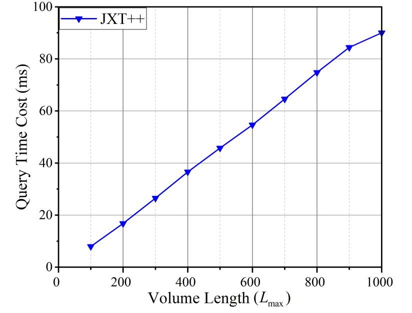

Figure 6: Query Time With Volumes

Table 3: Storage cost evaluation.

Scheme	$H(X)^6 = 16$	$H(X) = 14$	$H(X) = 12$
JXT [22]	55.1MB	53.6MB	53.2MB
JXT+	39.5MB	29.6MB	27.1MB
JXT++	43.7MB	22.5MB	17.2MB

one record identifier column). In total, the dataset involves 393,210 rows and 60 attribute columns. All experiments are conducted on the same machine equipped with Intel i5-11500 2.70GHz CPU and 16GB RAM. To ensure precise cost measurement, all experiments are executed on the same device for both the client and server. The reported running times are the average values derived from 1000 experiments. Our source code is available on GitHub[5].

We will provide a complete evaluation of setup and search costs in JXT, JXT+, and JXT++. Specifically, we first evaluate the time cost and storage size for generating an encrypted table with various numbers of join attributes. We also discuss the required storage size with different entropies of join-attribute. To achieve a reasonable comparison on join query, we first perform equi-join queries over two tables, covering all cases where the intermediate result from TSet *all* or *partly* match the query criteria. We further evaluate the time cost of equi-join query over multiple tables. Due to the original JXT and JXT+ support only two-table join, the corresponding variants (i.e., MJXT and MJXT+) are used to perform multiple-table join comparison in our experiment. To reflect the influence of result volume, we compare the query time cost of JXT++ based on different volume lengths L_{max}.

In the following experiments, the XSet is implemented by Bloom filter with approximate 10^{-12} false positive rate. The CSet in JXT+ is constructed as the multi-map, and in JXT++ is realized as the XOR filter whose storage capacity is $\lfloor 1.23n \rfloor + \beta$. According to [15, 37], to guarantee the probability of success and relatively less storage, the parameter β of the XOR filter is set to 2 in our construction.

6.3 Evaluation and Comparison

Storage Cost. We note that in JXT, JXT+, and JXT++, each table in the database is stored independently, and the total storage cost is essentially the sum of the size of all (encrypted) tables. Thus, we focus mainly on the storage overhead for each individual table. Note that the server-side storage overhead of JXT is dominated by TSet and XSet, while CSet is additionally required for both JXT+ and JXT++.

To provide an overall evaluation of storage cost, we choose 1/2/3/4/5 attributes from 10 attributes as join attributes, respectively. To facilitate comparison, all join attributes are high-entropy attribute columns, where the frequency of each join-attribute value is set to 2. As shown in Figure 1, our two protocols require less storage than that of JXT, even with additional data structure CSet. The main reason is the sizes of XSet in JXT and our schemes are mT, nT (best case), respectively, where m (*resp.* n) denotes the row (*resp.* column) number. Our two schemes require less storage cost in most cases. In particular, we perform the storage cost comparison of all the above schemes with different entropies of the join attribute. As indicated in Table 3, as the decrease of join-attribute entropy, the total storage cost is reduced. when the entropy of join attribute is set to 12, the storage cost of JXT is 53.2MB, while those of JXT+ and JXT++ are 27.1MB and 17.2MB, which brings a storage saving of 49% and 68%, respectively.

[5]https://github.com/CDSecLab/MJXT.
[6]$H(X)$ refers to the entropy of the distribution of join-attribute values X.

Setup Evaluation. Next, we discuss the setup efficiency for tables with 1/2/3/4/5 join attributes, chosen from a total of 10 attributes. The setup cost includes generating TSet and XSet for three schemes and additionally generating CSet for JXT+ and JXT++. Figure 4 illustrates that the setup cost for all three schemes increases with the number of join attributes. Specifically, JXT+ is less efficient than JXT due to the additional generation of CSet. JXT++ is the least efficient, as it further generates the XOR filter of CSet.

Query Evaluation. In the following, we investigate the query efficiency, which consists of the total time of generating the search token and decrypting search results on the client side, and performing search on the server side.

Firstly, we discuss the time cost of join query over two tables when the retrieved TSet entries (i.e., matched records in TSet) *all* satisfy the search condition. The number of retrieved TSet entries is set as $1,000/2,000/\ldots/10,000$ for each query, respectively. As depicted in Figure 2, the query time cost of all the mentioned schemes increases with the matched records in TSet growth. It can be observed that JXT+ and JXT++ are superior to JXT in search latency. Specifically, for the number of matched records is $1,000$, JXT takes $61ms$ to fetch $1,000 * 2$ identifiers (from two joined- tables) and decrypt all of them, while JXT+ takes $11.2ms$, a speedup of $5.5\times$, and JXT++ takes $9.7ms$, a speedup of $6.3\times$. For the number of matched records is $10,000$, JXT takes $4,651.5ms$ to fetch $10,000 * 2$ identifiers and decrypt all of them, while JXT+ takes $89.9ms$, a speedup of $51.7\times$, and JXT++ takes $85.7ms$, a speedup of $54.3\times$.

Next, we evaluate the query efficiency when the retrieved TSet entries *partially* satisfy the query. In detail, the number of retrieved entries from TSet is fixed as $1,000$, and the number of final result is $10\%/20\%/\ldots/100\%$ of retrieved TSet entries. As shown in Figure 3, JXT is minimally affected by the size of the final results. This is because the server-side cost depends only on the number of entries retrieved from TSet, while the client decrypts the reduced size of the final search results. Therefore, a smaller result size reduces the client's decryption time. In contrast, JXT+ and JXT++ become more efficient as the size of the final results decreases. The reason is that the server only retrieves the items in CSet satisfying the query (i.e., the final search results). Figure 3 shows that when the size of final results is 100, JXT takes $46.2ms$ to obtain the final results, while JXT+ takes only $1.4ms$, achieving a speedup of $33\times$. JXT++ is more efficient, taking just $0.9ms$, resulting in a speedup of $51.3\times$.

We now consider performing queries over 2/3/4/5/6 tables. In Figure 5, we evaluate the query time of MJXT, MJXT, and JXT++ under the condition that each table contains 1,000 matching identifiers. The results show that the query time for all three schemes increases as the number of joined tables grows, due to the additional computation and retrieval of more identifiers. Notably, JXT++ performs the best, despite offering the highest level of security. In addition, Figure 6 illustrates the impact of the volume length L_{max} on the query time of JXT++, where L_{max} captures the maximum occurrence number of all pairs (w, w^*). As the maximum volume length L_{max} increases, the query time of JXT++ grows linearly. This is because the server must handle and decrypt additional matching results, which arise from the increased dummy padding associated with the growth in L_{max}. In particular, when $L_{max} = 1000$, the query time for JXT++ on the two tables approaches $90ms$. Therefore, by setting L_{max} in join queries, we can achieve higher security while maintaining reasonable efficiency.

7 Conclusion

In this paper, we investigate secure join queries without join pre-computation in encrypted relational databases. We first present JXT+, a new equi-join query protocol over two tables without join pre-computation, which can support joins of attributes with different names and achieve better query efficiency. We then design the *first* equi-join query protocol across three or more tables, dubbed JXT++, which enjoys joins of arbitrary attributes even low-entropy attributes, while providing a tunable query complexity. Furthermore, we implement our two protocols and perform a complete comparison with the state-of-the-art JXT. Experimental results demonstrate that both of our proposed schemes are superior to JXT in terms of query and storage efficiency while achieving more powerful functionality.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. This work is supported by the National Key Research and Development Program of China (No. 2022YFB3102400), the National Natural Science Foundation of China (Nos. 62072357 and 62102313), the Key Research and Development Program of Shaanxi (No. 2022KWZ-01), and the Young Talent Fund of Association for Science and Technology in Shaanxi (No. 20230117).

References

[1] Ghous Amjad, Seny Kamara, and Tarik Moataz. 2023. Injection-Secure Structured and Searchable Symmetric Encryption. In *ASIACRYPT 2023 (LNCS, Vol. 14443)*. 232–262.
[2] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. Deterministic and Efficiently Searchable Encryption. In *CRYPTO 2007 (LNCS, Vol. 4622)*. 535–552.
[3] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2020. Revisiting Leakage Abuse Attacks. In *NDSS 2020*.
[4] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *CRYPTO 2011 (LNCS, Vol. 6841)*. 578–595.
[5] Raphael Bost. 2016. $\sum o\varphi o\varsigma$: Forward Secure Searchable Encryption. In *ACM 2016*. 1143–1154.
[6] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *ACM 2017*. 1465–1482.
[7] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and Space-Efficient Minimal Perfect Hash Functions. In *WADS 2007*. 139–150.
[8] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *ACM 2015*. 668–679.
[9] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS 2014*.
[10] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Catualin Roşu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO 2013 (LNCS, Vol. 8042)*. 353–373.
[11] David Cash, Ruth Ng, and Adam Rivkin. 2021. Improved Structured Encryption for SQL Databases via Hybrid Indexing. In *ACNS 2021*. 480–510.
[12] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *ASIACRYPT 2010 (LNCS, Vol. 6477)*. 577–594.
[13] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *ACM 2006*. 79–88.
[14] Marilyn George, Seny Kamara, and Tarik Moataz. 2021. Structured Encryption and Dynamic Leakage Suppression. In *EUROCRYPT 2021 (LNCS, Vol. 12698)*. 370–396.

[15] Thomas Mueller Graf and Daniel Lemire. 2020. Xor filters: Faster and Smaller Than Bloom and Cuckoo Filters. *ACM Journal of Experimental Algorithmics* 25 (2020), 1–16.
[16] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-Abuse Attacks against Order-Revealing Encryption. In *S&P 2017*. 655–672.
[17] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *ACM 2019*. 361–378.
[18] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *SIGMOD 2002*. 216–227.
[19] Florian Hahn, Nicolas Loza, and Florian Kerschbaum. 2019. Joins over Encrypted Data with Fine Granular Security. In *ICDE 2019*. 674–685.
[20] Alexander Hoover, Ruth Ng, Daren Khu, Yao'an Li, Joelle Lim, Derrick Ng, Jed Lim, and Yiyang Song. 2024. Leakage-Abuse Attacks Against Structured Encryption for SQL. In *USENIX Security 2024*.
[21] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS 2012*.
[22] Charanjit Jutla and Sikhar Patranabis. 2022. Efficient Searchable Symmetric Encryption for Join Queries. In *ASIACRYPT 2022*. 304–333.
[23] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Thomas Schneider, Amos Treiber, and Michael Yonli. 2022. SoK: Cryptanalysis of Encrypted Search with LEAKER - A framework for LEakage AttacK Evaluation on Real-world data. In *EuroS&P 2022*. 90–108.
[24] Seny Kamara and Tarik Moataz. 2017. Boolean Searchable Symmetric Encryption with Worst-Case Sub-linear Complexity. In *EUROCRYPT 2017 (LNCS, Vol. 10212)*. 94–124.
[25] Seny Kamara and Tarik Moataz. 2018. SQL on Structurally-Encrypted Databases. In *ASIACRYPT 2018 (LNCS, Vol. 11272)*. 149–180.
[26] Seny Kamara and Tarik Moataz. 2019. Computationally Volume-Hiding Structured Encryption. In *EUROCRYPT 2019 (LNCS, Vol. 11477)*. 183–213.
[27] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shi-Feng Sun, Dongxi Liu, and Cong Zuo. 2018. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In *CCS 2018*. 745–762.
[28] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. 1996. A Family of Perfect Hashing Methods. *Comput. J.* (1996), 547–554.
[29] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS 2015*. 644–655.
[30] Simon Oya and Florian Kerschbaum. 2021. Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption. In *USENIX Security 2021*. 127–142.
[31] Raluca A. Popa, Frank H. Li, and Nickolai Zeldovich. 2013. An Ideal-Security Protocol for Order-Preserving Encoding. In *S&P 2013*. 463–477.
[32] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP 2011*. 85–100.
[33] David Pouliot and Charles V. Wright. 2016. The Shadow Nemesis: Inference Attacks on Efficiently Deployable, Efficiently Searchable Encryption. In *CCS 2016*. 1341–1352.
[34] Masoumeh Shafieinejad, Suraj Gupta, Jin Yang Liu, Koray Karabina, and Florian Kerschbaum. 2022. Equi-joins over Encrypted Data for Series of Queries. In *ICDE 2022*. 1635–1648.
[35] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *S&P 2000*. 44–55.
[36] Shifeng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. 2018. Practical Backward-Secure Searchable Encryption from Symmetric Puncturable Encryption. In *CCS 2018*. 763–780.
[37] Jianfeng Wang, Shi-Feng Sun, Tianci Li, Saiyu Qi, and Xiaofeng Chen. 2022. Practical volume-hiding encrypted multi-maps with optimal overhead and beyond. In *CCS 2022*. 2825–2839.
[38] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All your Queries are Belong to us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX Security 2016*. 707–720.

A Proof of Non-adaptive Security for JXT+

We prove this theorem by a sequence of games $G_0, \ldots, G_7$. Each game begins with an $\mathsf{Initialize}$ routine that takes $(\mathsf{DB}, \mathbf{q})$ as input from the adversary $\mathcal{A}$, who finally outputs a bit as the game's output based on the routine's result. The first game G_0 has the identical distribution to "real-world" game $\mathbf{Real}_{\mathcal{A}}^{\mathsf{JXT+}}(\lambda)$ and the final one G_7 can be simulated with leakage profile instead of the actual $(\mathsf{DB}, \mathbf{q})$. For simplicity, a sequence of Q non-adaptive join queries across two tables is denoted as $\mathbf{q} = (\mathbf{t}_1, \mathbf{t}_2, \mathbf{w}_1, \mathbf{w}_2, \mathbf{attr}^*_{\mathbf{t}_1}, \mathbf{attr}^*_{\mathbf{t}_2})$, where each query $\mathbf{q}[i] = (\mathbf{t}_1[i], \mathbf{t}_2[i], \mathbf{w}_1[i], \mathbf{w}_2[i], \mathbf{attr}^*_{\mathbf{t}_1}[i], \mathbf{attr}^*_{\mathbf{t}_2}[i])$ is a two-table join query "*Select* inds *From* $\mathsf{Tab}_{\mathbf{t}_1[i]}$ and $\mathsf{Tab}_{\mathbf{t}_2[i]}$ *Join on* $\mathbf{attr}^*_{\mathbf{t}_1}[i] = \mathbf{attr}^*_{\mathbf{t}_2}[i]$ *Where* $\mathbf{w}_1[i] \wedge \mathbf{w}_2[i]$", for $1 \le i \le Q$. In the following, we will provide the details of each game and formally prove the computational indistinguishability between them.

Game G_0: This game is instantiated based on $\mathbf{Real}_{\mathcal{A}}^{\mathsf{JXT+}}(\lambda)$ with minor differences. Specifically, this game simulates the encrypted database EDB and the transcript $\mathbf{Tr}$ by running $\mathsf{Initialize}$ based on $(\mathsf{DB}, \mathbf{q})$ selected by the adversary $\mathcal{A}$. (1) $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$ is simulated exactly as $\mathbf{EDBSetup}(\mathsf{DB})$ described in the real game. (2) $\mathbf{Tr}[i] = \big((\mathsf{STags}[i], \mathsf{xjointoken}_1[i], \mathsf{xjointoken}_2[i]), \mathsf{ResCT}[i], \mathsf{ResInd}[i]\big)$ for each query i $(1 \le i \le Q)$ is generated as in the real game, except that the plaintext result ResInd is derived from DB rather than from decrypting the encrypted search result ResCT. Thus, assuming no false positives, the distribution of G_0 matches $\mathbf{Real}_{\mathcal{A}}^{\mathsf{JXT+}}(\lambda)$, so we have

$$\Pr[G_0 = 1] \le \Pr[\mathbf{Real}_{\mathcal{A}}^{\mathsf{JXT+}}(\lambda) = 1] + \mathsf{negl}(\lambda).$$

Game G_1: In this game, the PRFs $F(K_z, \cdot)$, $F(K_w, \cdot)$, $F(K_r, \cdot)$, and $F(K_{\text{enc}}, \cdot)$ are replaced with the independent random functions or selections with the appropriate domain and range. Therefore, there exists an efficient adversary $\mathcal{B}_1$ such that

$$\big|\Pr[G_1 = 1] - \Pr[G_0 = 1]\big| \le 4 \cdot Adv_{F,\mathcal{B}_1}^{\mathsf{PRF}}(\lambda).$$

Game G_2: This game is identical to game G_1, except that the ciphertext ct is generated with encryption of a string 0^λ instead of the actual record identifier used in the real game. Given that the number of encryption operations is polynomial, i.e., $poly(\lambda)$, an efficient adversary $\mathcal{B}_2$ can be constructed such that

$$\big|\Pr[G_2 = 1] - \Pr[G_1 = 1]\big| \le poly(\lambda) \cdot Adv_{\mathsf{SE},\mathcal{B}_2}^{\mathsf{IND-CPA}}(\lambda).$$

Game G_3: This game differs in the computation process but is distributionally identical to G_2. Specifically, it precomputes all possible values for generating XSet, CSet, $\mathsf{xjointoken}_1$, and $\mathsf{xjointoken}_2$ and collects them in arrays H and Y. These include values present in XSet and CSet, all possible values for membership testing in XSet, and values for $\mathsf{xjointoken}_1$ and $\mathsf{xjointoken}_2$ that do not correspond to any potential matches. Then the corresponding values in XSet and CSet are selected from H, while $\mathsf{xjointoken}_1$ and $\mathsf{xjointoken}_2$ are computed by choosing values from either H or Y. Thus, we have

$$\Pr[G_3 = 1] = \Pr[G_2 = 1].$$

Game G_4: This game is exactly like G_3 except that each y in T and the values in H and Y arrays are choosen at randomly from $\{0,1\}^\lambda$. We claim that

$$\Pr[G_4 = 1] = \Pr[G_3 = 1].$$

Game G_5: In this game, instead invoking the real TSetSetup and TSetGetTag to compute TSet and STags, they are generated by constructing a non-adaptive simulator $\mathcal{S}_\mathsf{T}$ on input $\mathcal{L}_\mathsf{T}(\mathsf{DB}, \mathbf{t}_1, \mathbf{w}_1, \mathbf{attr}^*_{\mathbf{t}_1})$ and $\mathsf{T}_{\mathbf{t}_1}[\mathbf{w}_1 || \mathbf{attr}^*_{\mathbf{t}_1}]$. Therefore, a non-adaptive secure TSet instantiation ensures the existence of $\mathcal{S}_\mathsf{T}$ such that the following holds:

$$\big|\Pr[G_5 = 1] - \Pr[G_4 = 1]\big| \le Adv_{\mathcal{B}_5,\mathcal{S}_\mathsf{T}}^{\mathsf{TSet}}(\lambda).$$

Game G_6: This game modifies the access to the H and Y arrays to ensure compatibility with the final simulator using the leakage

profile. When accessing H at $(w, attr^*, w^*)$ or Y at (w, u, c), the game first checks if that index will be accessed again. If so, it uses the existing value; if not, it substitutes a random value. Since those indices won't be reused, this change doesn't affect the distribution of the game. Notably, the Y array is only accessed once during transcript generation and can be replaced with random values, so leading to the removal of the Y array from this game.

Next, we describe the way of accessing the H array. The H array is accessed only during two processes: (1) XSet&CSetSetup: the generation of XSet and CSet, and (2) GenTrans: the generation of the transcript, specifically for $\mathsf{xjointoken}_1$ and $\mathsf{xjointoken}_2$.

The XSet&CSetSetup routine does not repeatedly access H, but some of them might be accessed by GenTrans. We observe that GenTrans reads indices such that $(\mathbf{w}_1[i] = w \vee \mathbf{w}_2[i] = w) \wedge w^* \in \{\text{val}^* : (\text{ind}_1, \mathbf{attr}^*_{\mathbf{t}_1}[i], \text{val}^*) \in \mathsf{Tab}_{\mathbf{t}_1[i]} \wedge (\text{ind}_2, \mathbf{attr}^*_{\mathbf{t}_2}[i], \text{val}^*) \in \mathsf{Tab}_{\mathbf{t}_2[i]} \wedge (\text{ind}_1, \mathbf{w}_1[i]) \in \mathsf{Tab}_{\mathbf{t}_1[i]} \wedge (\text{ind}_2, \mathbf{w}_2[i]) \in \mathsf{Tab}_{\mathbf{t}_2[i]}\}$ for a given query i. Therefore, we use the corresponding values from H at these indices in the generation of XSet and CSet and replace the others with random values.

In the GenTrans routine, we need to check for repeated index accesses, including positions read by XSet&CSetSetup or revisited by GenTrans itself. First, we determine which indices correspond to the final matches and access values from H at these positions. These values are clearly among those accessed by XSet&CSetSetup, so GenTrans tests for the same condition. Then, we observe that some positions are accessed twice in two different GenTrans calls. For some $i \neq j$, w^* is an element of both $\{\text{val}^* : (\text{ind}, \mathbf{w}_1[j]) \in \mathsf{Tab}_{\mathbf{t}_1[j]} \wedge (\text{ind}, \mathbf{attr}^*_{\mathbf{t}_1}[j], \text{val}^*) \in \mathsf{Tab}_{\mathbf{t}_1[j]}\}$ and $\{\text{val}^* : (\text{ind}, \mathbf{w}_1[i]) \in \mathsf{Tab}_{\mathbf{t}_1[i]} \wedge (\text{ind}, \mathbf{attr}^*_{\mathbf{t}_1}[i], \text{val}^*) \in \mathsf{Tab}_{\mathbf{t}_1[i]}\}$. For these repeated accesses determined by w^*, we use the values from H. In all other cases, GenTrans replaces the H accesses with random values.

Based on the above observations and discussion, we have

$$\Pr[\mathsf{G}_6 = 1] = \Pr[\mathsf{G}_5 = 1].$$

Simulator: This game constructs a simulator $\mathcal{S}$ and takes the leakage profile $\mathcal{L}(\mathsf{DB}, \mathbf{q})$ as an input, which consists of $\mathcal{L}_{\mathsf{JXT+}}(\mathsf{DB}, \mathbf{q})$, $\mathcal{L}_{\mathsf{T}}(\mathsf{DB}, \mathbf{t}_1, \mathbf{w}_1, \mathbf{attr}^*_{\mathbf{t}_1})$, and $\mathsf{T}_{\mathbf{t}_1}[\mathbf{w}_1 || \mathbf{attr}^*_{\mathbf{t}_1}]$. It finally outputs the simulated $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$ and transcripts $\mathbf{Tr} = ((\mathsf{STags}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2), \mathsf{ResCT}, \mathsf{ResInd})$, which has same distribution as G_6.

<u>TSet Simulation:</u> The non-adaptive simulator $\mathcal{S}_{\mathsf{T}}$ is invoked to generate TSet and STags based on $\mathcal{L}_{\mathsf{T}}(\mathsf{DB}, \mathbf{t}_1, \mathbf{w}_1, \mathbf{attr}^*_{\mathbf{t}_1})$, and $\mathsf{T}_{\mathbf{t}_1}[\mathbf{w}_1 || \mathbf{attr}^*_{\mathbf{t}_1}]$, which has the same distribution as in G_6. Note that $\mathsf{T}_{\mathbf{t}_1}[\mathbf{w}_1 || \mathbf{attr}^*_{\mathbf{t}_1}]$ can be computed by randomly selecting y from $\{0, 1\}^\lambda$.

Next, we utilize the leakage $\mathcal{L}_{\mathsf{JXT+}}(\mathsf{DB}, \mathbf{q}) = (\mathsf{n}, \mathsf{RP}, \mathsf{EP}_1, \mathsf{EP}_2, \mathsf{SP}_1, \mathsf{JD})$ to simulate XSet and CSet, and then produce the transcript $\mathbf{Tr}$. Before this, the simulator $\mathcal{S}$ preprocesses some leakage to facilitate their use. Specifically, the simulator $\mathcal{S}$ first generates two leakage variants $\overline{\mathsf{RP}}$ and $\overline{\mathsf{EP}}_2$ based on the result pattern leakage RP and the equality pattern leakage EP_2, respectively.

- For each join query $\mathbf{q}[i]$ $(i \in [Q])$, $\overline{\mathsf{RP}}[i] = \{\text{encode}(\text{val}^*) : \text{ind} \in \mathsf{RP}[i] \wedge (\text{ind}, \mathbf{attr}^*_{\mathbf{t}_1}[i], \text{val}^*) \in \mathsf{Tab}_{\mathbf{t}_1[i]}\}$ is a multi-set of join-attribute values (in the randomized encoding version) for the record identifiers appearing in the final result $\mathsf{RP}[i]$ in table $\mathsf{Tab}_{\mathbf{t}_1[i]}$.
- $\overline{\mathsf{EP}}_2$ is the restricted equality pattern of $\mathbf{w}_2$ and is also represented by an integer vector, which can be derived from EP_2 and JD. Specifically, we define $\overline{\mathsf{EP}}_2[i] = \overline{\mathsf{EP}}_2[j]$ iff $\mathsf{EP}_2[i] = \mathsf{EP}_2[j]$ and $\mathsf{JD}[i] \cap \mathsf{JD}[j] \neq \emptyset$ for $i, j \in [Q]$.

<u>XSet and CSet Simulation:</u> The simulator $\mathcal{S}$ precomputes the H array by combining EP_1 with JD and $\overline{\mathsf{EP}}_2$ with JD, encompassing all values $H[w, w^*]$ that may or may not be accessed, where w is from EP and w^* is from JD. Then the simulator uses the variant $\overline{\mathsf{RP}}$ of result pattern leakage to select the corresponding values $H[w, w^*]$ that may appear in the result set and store them into XSet. Simultaneously, the simulator inserts $(H[w, w^*], ct)$ into CSet, where ct is the ciphertext of 0^λ.

<u>Transcript Simulation:</u> The simulator $\mathcal{S}$ computes the transcript $\mathbf{Tr}[i] = ((\mathsf{STags}[i], \mathsf{xjointoken}_1[i], \mathsf{xjointoken}_2[i]), \mathsf{ResCT}[i], \mathsf{ResInd}[i])$, for $1 \leq i \leq Q$. Specifically, STags is simulated within TSet simulation, $\mathsf{ResInd}[i]$ can be derived from $\mathsf{RP}[i]$, and $\mathsf{ResCT}[i]$ is simulated by the server search subroutine as described in real game. Furthermore, we describe how to compute $\mathsf{xjointoken}_1$ and $\mathsf{xjointoken}_2$. $\mathcal{S}$ collects the encoding of the join-attribute values for a given i as $R \leftarrow \overline{\mathsf{RP}}[i] \cup \bigcup_{j=1}^{Q} \mathsf{JD}[j] \cap \mathsf{JD}[i]$ and puts them in canonical order as $(\bar{w}^*_1, \bar{w}^*_2, \ldots, \bar{w}^*_{|R|}) \leftarrow R$. Since each index in R belongs to $\mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_{t_1}}(w_1)$, it follows $|R| \leq \mathsf{SP}_1[i]$. $\mathcal{S}$ pads $|R|$ up to $\mathsf{SP}_1[i]$ by setting $\bar{w}^*_k \leftarrow \perp$ for $k = |R| + 1, \ldots, \mathsf{SP}_1[i]$. The simulator then determines whether xjointoken is obtained from H array or random values. Specifically, for $c = 1, \ldots, \mathsf{SP}_1[i]$, if $\bar{w}^*_c \neq \perp$, then $\mathsf{xjointoken}_1[c]$ and $\mathsf{xjointoken}_2[c]$ are computed by using $H[w, w^*] \oplus y$; otherwise, they are assigned random values from $\{0, 1\}^\lambda$. In addition, for $c = \mathsf{SP}_1[i] + 1, \ldots, T_{\max}$, we also select random values for $\mathsf{xjointoken}_1[c]$ and $\mathsf{xjointoken}_2[c]$. Note that $T_{\max}$ is an upper bound to fix the number of xjointoken for simplicity. Thus, the distributions of $\mathsf{xjointoken}_1$ and $\mathsf{xjointoken}_2$ generated by the simulator are the same as those produced in G_6.

In summary, the simulator $\mathcal{S}$ with the leakage $\mathcal{L}(\mathsf{DB}, \mathbf{q})$ generates a distribution identical to that of G_6, so we complete the proof.

Algorithm 4 Game G_0

Initialize$(\mathsf{DB}, \mathsf{t}_1, \mathsf{t}_2, \mathsf{w}_1, \mathsf{w}_2, \mathsf{attr}^*_{\mathsf{t}_1}, \mathsf{attr}^*_{\mathsf{t}_2})$

1: $K_z, K_w, K_r, K_{\text{enc}} \xleftarrow{\$} \{0,1\}^\lambda$, $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]} \leftarrow \mathsf{DB}$
2: **for** $i = 1$ to N **do**
3: $\mathsf{T}_i \leftarrow$ empty array
4: **for** $w \in \mathsf{W}_i$ **do**
5: $\{(\overline{\text{ind}}_1, \{attr^*_t, w^*_t\}_{t\in[T]}), \ldots, (\overline{\text{ind}}_{T_w}, \{attr^*_t, w^*_t\}_{t\in[T]})\} \leftarrow \mathsf{DB}^{\text{Join}}_{\mathsf{Tab}_i}(w)$
6: **for** $c = 1, \ldots, T_w$ **do**
7: $Z_{\text{cnt}} \leftarrow F(K_z, w||c)$
8: **for** $t \in [T]$ **do**
9: $y \leftarrow F(K_w, w^*_t) \oplus Z_{\text{cnt}}$, $\mathsf{T}_i[w||attr^*_t][c] \leftarrow y$
10: **end for**
11: **end for**
12: **end for**
13: **end for**
14: $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathsf{T}_1||...||\mathsf{T}_N)$
15: **for** $i = 1$ to Q **do** $\mathsf{STags}[i] \leftarrow \mathsf{TSetGetTag}(K_T, (\mathsf{t}_1[i], \mathsf{w}_1[i]||\mathsf{attr}^*_{\mathsf{t}_1}[i]))$
16: **end for**
17: $(\mathsf{XSet}, \mathsf{CSet}) \leftarrow$ **XSet&CSetSetup**$(K_z, K_w, K_r, K_{\text{enc}}, \mathsf{DB})$
18: $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$
19: **for** $i = 1$ to Q **do**
20: $\mathsf{Tr}[i] \leftarrow$ **GenTrans**$(\mathsf{EDB}, K_z, K_r, \mathsf{t}_1[i], \mathsf{t}_2[i], \mathsf{w}_1[i], \mathsf{w}_2[i], \mathsf{attr}^*_{\mathsf{t}_1}[i], \mathsf{attr}^*_{\mathsf{t}_2}[i], \mathsf{STags}[i])$
21: **end for**
22: **return** $(\mathsf{EDB}, \mathsf{Tr})$

XSet&CSetSetup$(K_z, K_w, K_r, K_{\text{enc}}, \mathsf{DB})$

1: $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]} \leftarrow \mathsf{DB}$
2: **for** $i = 1$ to N **do**
3: $\mathsf{XSet}[i] \leftarrow \emptyset$, $\mathsf{CSet}[i] \leftarrow$ empty multi-map
4: **for** $w \in \mathsf{W}_i$ and $(\text{ind}, \{attr^*_t, w^*_t\}_{t\in[T]}) \in \mathsf{DB}^{\text{Join}}_{\mathsf{Tab}_i}(w)$ **do**
5: $Z_0 \leftarrow F(K_z, w||0)$
6: $\mathsf{xtag} \leftarrow F(K_r, attr^*_t) \oplus F(K_w, w^*_t) \oplus Z_0$
7: $\mathsf{XSet}[i] \leftarrow \mathsf{XSet}[i] \cup \{\mathsf{xtag}\}$
8: $K_{\text{enc},w} \leftarrow F(K_{\text{enc}}, w)$
9: $ct \leftarrow \mathsf{Enc}(K_{\text{enc},w}, \text{ind})$
10: $\mathsf{CSet}[i] \leftarrow \mathsf{CSet}[i].\mathsf{Put}(\mathsf{xtag}, ct)$
11: **end for**
12: **end for**
13: **return** $(\mathsf{XSet}, \mathsf{CSet})$

GenTrans$(\mathsf{EDB}, K_z, K_r, t_1, t_2, w_1, w_2, attr^*_{t_1}, attr^*_{t_2}, \mathsf{stag})$

1: **for** $c = 1, \ldots, T_{\max}$ **do**
2: $\mathsf{xjointoken}_1[c] \leftarrow F(K_z, w_1||0) \oplus F(K_r, attr^*_{t_1}) \oplus F(K_z, w_1||c)$
3: $\mathsf{xjointoken}_2[c] \leftarrow F(K_z, w_2||0) \oplus F(K_r, attr^*_{t_2}) \oplus F(K_z, w_1||c)$
4: **end for**
5: $\mathsf{ResCT} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (t_1, t_2, \mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2))$
6: $q \leftarrow (t_1, t_2, w_1, w_2, attr^*_{t_1}, attr^*_{t_2})$
7: $\mathsf{ResInd} \leftarrow \mathsf{DB}(q)$
8: **return** $((\mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2), \mathsf{ResCT}, \mathsf{ResInd})$

Algorithm 5 Game G_1 and Game G_2

Initialize$(\mathsf{DB}, \mathsf{t}_1, \mathsf{t}_2, \mathsf{w}_1, \mathsf{w}_2, \mathsf{attr}^*_{\mathsf{t}_1}, \mathsf{attr}^*_{\mathsf{t}_2})$

1: $f_z, f_w, f_r \xleftarrow{\$} \mathsf{Fun}(\{0,1\}^\lambda)$, $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]} \leftarrow \mathsf{DB}$
2: **for** $i = 1$ to N **do**
3: $\mathsf{T}_i \leftarrow$ empty array
4: **for** $w \in \mathsf{W}_i$ **do**
5: $\{(\overline{\text{ind}}_1, \{attr^*_t, w^*_t\}_{t\in[T]}), \ldots, (\overline{\text{ind}}_{T_w}, \{attr^*_t, w^*_t\}_{t\in[T]})\} \leftarrow \mathsf{DB}^{\text{Join}}_{\mathsf{Tab}_i}(w)$
6: **for** $c = 1, \ldots, T_w$ **do**
7: $Z_{\text{cnt}} \leftarrow f_z(w||c)$
8: **for** $t \in [T]$ **do**
9: $y \leftarrow f_w(w^*_t) \oplus Z_{\text{cnt}}$, $\mathsf{T}_i[w||attr^*_t][c] \leftarrow y$
10: **end for**
11: **end for**
12: **end for**
13: **end for**
14: $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathsf{T}_1||...||\mathsf{T}_N)$
15: **for** $i = 1$ to Q **do** $\mathsf{STags}[i] \leftarrow \mathsf{TSetGetTag}(K_T, (\mathsf{t}_1[i], \mathsf{w}_1[i]||\mathsf{attr}^*_{\mathsf{t}_1}[i]))$
16: **end for**
17: $(\mathsf{XSet}, \mathsf{CSet}) \leftarrow$ **XSet&CSetSetup**$(f_z, f_w, f_r, \mathsf{DB})$
18: $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$
19: **for** $i = 1$ to Q **do**
20: $\mathsf{Tr}[i] \leftarrow$ **GenTrans**$(\mathsf{EDB}, f_z, f_r, \mathsf{t}_1[i], \mathsf{t}_2[i], \mathsf{w}_1[i], \mathsf{w}_2[i], \mathsf{attr}^*_{\mathsf{t}_1}[i], \mathsf{attr}^*_{\mathsf{t}_2}[i], \mathsf{STags}[i])$
21: **end for**
22: **return** $(\mathsf{EDB}, \mathsf{Tr})$

XSet&CSetSetup$(f_z, f_w, f_r, \mathsf{DB})$

1: $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]} \leftarrow \mathsf{DB}$
2: **for** $i = 1$ to N **do**
3: $\mathsf{XSet}[i] \leftarrow \emptyset$, $\mathsf{CSet}[i] \leftarrow$ empty multi-map
4: **for** $w \in \mathsf{W}_i$ and $(\text{ind}, \{attr^*_t, w^*_t\}_{t\in[T]}) \in \mathsf{DB}^{\text{Join}}_{\mathsf{Tab}_i}(w)$ **do**
5: $Z_0 \leftarrow f_z(w||0)$
6: $\mathsf{xtag} \leftarrow f_r(attr^*_t) \oplus f_w(w^*_t) \oplus Z_0$
7: $\mathsf{XSet}[i] \leftarrow \mathsf{XSet}[i] \cup \{\mathsf{xtag}\}$
8: $K_{\text{enc},w} \xleftarrow{\$} \{0,1\}^\lambda$
9: $ct \leftarrow \mathsf{Enc}(K_{\text{enc},w}, \text{ind})$
10: $ct \leftarrow \mathsf{Enc}(K_{\text{enc},w}, 0^\lambda)$
11: $\mathsf{CSet}[i] \leftarrow \mathsf{CSet}[i].\mathsf{Put}(\mathsf{xtag}, ct)$
12: **end for**
13: **end for**
14: **return** $(\mathsf{XSet}, \mathsf{CSet})$

GenTrans$(\mathsf{EDB}, f_z, f_r, t_1, t_2, w_1, w_2, attr^*_{t_1}, attr^*_{t_2}, \mathsf{stag})$

1: **for** $c = 1, \ldots, T_{\max}$ **do**
2: $\mathsf{xjointoken}_1[c] \leftarrow f_z(w_1||0) \oplus f_r(attr^*_{t_1}) \oplus f_z(w_1||c)$
3: $\mathsf{xjointoken}_2[c] \leftarrow f_z(w_2||0) \oplus f_r(attr^*_{t_2}) \oplus f_z(w_1||c)$
4: **end for**
5: $\mathsf{ResCT} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (t_1, t_2, \mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2))$
6: $q \leftarrow (t_1, t_2, w_1, w_2, attr^*_{t_1}, attr^*_{t_2})$
7: $\mathsf{ResInd} \leftarrow \mathsf{DB}(q)$
8: **return** $((\mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2), \mathsf{ResCT}, \mathsf{ResInd})$

Algorithm 6 Game G_3 and Game G_4

Initialize$(\mathsf{DB}, \mathbf{t}_1, \mathbf{t}_2, \mathbf{w}_1, \mathbf{w}_2, \mathbf{attr}^*_{\mathbf{t}_1}, \mathbf{attr}^*_{\mathbf{t}_2})$

1: $f_z, f_w, f_r \xleftarrow{\$} \mathsf{Fun}(\{0,1\}^\lambda)$, $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]} \leftarrow \mathsf{DB}$
2: **for** $i = 1$ to N **do**
3: **for** $w \in \mathsf{W}_i$ and $attr^* \in \mathsf{Tab}_i$ and $w^* \in \mathsf{DB}$ **do**
4: $Z_0 \leftarrow f_z(w||0)$
5: $H[w, attr^*, w^*] \leftarrow f_r(attr^*) \oplus f_w(w^*) \oplus Z_0$
6: $H[w, attr^*, w^*] \xleftarrow{\$} \{0,1\}^\lambda$
7: **end for**
8: **end for**
9: **for** $i = 1$ to N **do**
10: $\mathsf{T}_i \leftarrow$ empty array
11: **for** $w \in \mathsf{W}_i$ **do**
12: $\{(\overline{\mathsf{ind}}_1, \{attr^*_t, w^*_t\}_{t\in[T]}), \ldots, (\overline{\mathsf{ind}}_{T_w}, \{attr^*_t, w^*_t\}_{t\in[T]})\} \leftarrow \mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(w)$
13: **for** $c = 1, \ldots, T_w$ **do**
14: $Z_{\mathrm{cnt}} \leftarrow f_z(w||c)$
15: **for** $t \in [T]$ **do**
16: $y \leftarrow f_w(w^*_t) \oplus Z_{\mathrm{cnt}}$
17: $y \xleftarrow{\$} \{0,1\}^\lambda$
18: $\mathsf{T}_i[w||attr^*_t][c] \leftarrow y$
19: **end for**
20: **end for**
21: **for** $u \in w \cup \bigcup_{j\in[N]\setminus\{i\}} \mathsf{W}_j$ **do**
22: **for** $c = T_w + 1, \ldots, T_{\max}$ **do**
23: **for** $t \in [T]$ **do**
24: $Y[w, u, c] \leftarrow f_z(w||c) \oplus f_z(u||0)$
25: $Y[w, u, c] \xleftarrow{\$} \{0,1\}^\lambda$
26: **end for**
27: **end for**
28: **end for**
29: **end for**
30: **end for**
31: $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathsf{T}_1||\ldots||\mathsf{T}_N)$
32: **for** $i = 1$ to Q **do**
33: $\mathsf{STags}[i] \leftarrow \mathsf{TSetGetTag}(K_T, (\mathbf{t}_1[i], \mathbf{w}_1[i]||\mathbf{attr}^*_{\mathbf{t}_1}[i]))$
34: **end for**
35: $(\mathsf{XSet}, \mathsf{CSet}) \leftarrow$ **XSet&CSetSetup**(DB, H)
36: $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$
37: **for** $i = 1$ to Q **do**
38: $\mathbf{Tr}[i] \leftarrow$ **GenTrans**$(\mathsf{DB}, \mathsf{EDB}, f_r, H, Y, \mathbf{t}_1[i], \mathbf{t}_2[i], \mathbf{w}_1[i], \mathbf{w}_2[i], \mathbf{attr}^*_{\mathbf{t}_1}[i], \mathbf{attr}^*_{\mathbf{t}_2}[i], \mathsf{STags}[i])$
39: **end for**
40: **return** $(\mathsf{EDB}, \mathbf{Tr})$

XSet&CSetSetup(DB, H)

1: $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]} \leftarrow \mathsf{DB}$
2: **for** $i = 1$ to N **do**
3: $\mathsf{XSet}[i] \leftarrow \emptyset$, $\mathsf{CSet}[i] \leftarrow$ empty multi-map
4: **for** $w \in \mathsf{W}_i$ and $(\mathsf{ind}, \{attr^*_t, w^*_t\}_{t\in[T]}) \in \mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(w)$ **do**
5: $\mathsf{XSet}[i] \leftarrow \mathsf{XSet}[i] \cup \{H[w, attr^*_t, w^*_t]\}$
6: $K_{\mathrm{enc},w} \xleftarrow{\$} \{0,1\}^\lambda$
7: $ct \leftarrow \mathsf{Enc}(K_{\mathrm{enc},w}, 0^\lambda)$
8: $\mathsf{CSet}[i] \leftarrow \mathsf{CSet}[i].\mathsf{Put}(H[w, attr^*_t, w^*_t], ct)$
9: **end for**
10: **end for**
11: **return** $(\mathsf{XSet}, \mathsf{CSet})$

GenTrans$(\mathsf{DB}, \mathsf{EDB}, f_r, H, Y, t_1, t_2, w_1, w_2, attr^*_{t_1}, attr^*_{t_2}, \mathsf{stag})$

1: $\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$
2: $\{(\overline{\mathsf{ind}}_1, \{attr^*_t, w^*_t\}_{t\in[T]}), \ldots, (\overline{\mathsf{ind}}_{T_{w_1}}, \{attr^*_t, w^*_t\}_{t\in[T]})\} \leftarrow \mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_{t_1}}(w_1)$
3: **for** $c = 1, \ldots, T_{w_1}$ **do**
4: $y \leftarrow \mathbf{t}[c]$
5: $\mathsf{xjointoken}_1[c] \leftarrow H[w_1, attr^*_{t_1}, w^*_t] \oplus y$
6: $\mathsf{xjointoken}_2[c] \leftarrow H[w_2, attr^*_{t_2}, w^*_t] \oplus y$
7: **end for**
8: Note that w^*_t is determined by c and $attr^*_{t_1}$ using $\mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_{t_1}}(w_1)$.
9: **for** $c = T_{w_1} + 1, \ldots, T_{\max}$ **do**
10: $\mathsf{xjointoken}_1[c] \leftarrow Y[w_1, w_1, c] \oplus f_r(attr^*_{t_1})$
11: $\mathsf{xjointoken}_2[c] \leftarrow Y[w_1, w_2, c] \oplus f_r(attr^*_{t_2})$
12: **end for**
13: $\mathsf{ResCT} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (t_1, t_2, \mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2))$
14: $q \leftarrow (t_1, t_2, w_1, w_2, attr^*_{t_1}, attr^*_{t_2})$
15: $\mathsf{ResInd} \leftarrow \mathsf{DB}(q)$
16: **return** $((\mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2), \mathsf{ResCT}, \mathsf{ResInd})$

Algorithm 7 Game G_5 and Game G_6

Initialize$(\mathsf{DB}, \mathbf{t}_1, \mathbf{t}_2, \mathbf{w}_1, \mathbf{w}_2, \mathbf{attr}^*_{\mathbf{t}_1}, \mathbf{attr}^*_{\mathbf{t}_2})$

1: $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i \in [N]} \leftarrow \mathsf{DB}$
2: **for** $i = 1$ to N **do**
3:   **for** $w \in \mathsf{W}_i$ and $attr^* \in \mathsf{Tab}_i$ and $w^* \in \mathsf{DB}$ **do**
4:     $H[w, attr^*, w^*] \xleftarrow{\$} \{0,1\}^\lambda$
5:   **end for**
6: **end for**
7: **for** $i = 1$ to N **do**
8:   $\mathsf{T}_i \leftarrow$ empty array
9:   **for** $w \in \mathsf{W}_i$ **do**
10:     **for** $c = 1, \ldots, T_w$ **do**
11:       **for** $t \in [T]$ **do**
12:         $y \xleftarrow{\$} \{0,1\}^\lambda$
13:         $\mathsf{T}_i[w||t][c] \leftarrow y$
14:       **end for**
15:     **end for**
16:   **end for**
17: **end for**
18: $(\mathsf{TSet}, \mathsf{STags}) \leftarrow \mathcal{S}_\mathsf{T}(\mathcal{L}_\mathsf{T}(\mathsf{DB}, \mathbf{t}_1, \mathbf{w}_1, \mathbf{attr}^*_{\mathbf{t}_1}), \mathsf{T}_{\mathbf{t}_1}[\mathbf{w}_1||\mathbf{attr}^*_{\mathbf{t}_1}])$
19: $(\mathsf{XSet}, \mathsf{CSet}) \leftarrow$ **XSet&CSetSetup**(DB, H)
20: $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$
21: **for** $i = 1$ to Q **do**
22:   $\mathbf{Tr}[i] \leftarrow$ **GenTrans**$(\mathsf{DB}, \mathsf{EDB}, H, \mathbf{t}_1[i], \mathbf{t}_2[i], \mathbf{w}_1[i], \mathbf{w}_2[i], \mathbf{attr}^*_{\mathbf{t}_1}[i], \mathbf{attr}^*_{\mathbf{t}_2}[i], \mathsf{STags}[i])$
23: **end for**
24: **return** $(\mathsf{EDB}, \mathbf{Tr})$

XSet&CSetSetup(DB, H)

1: $\{\mathsf{Tab}_i, \mathsf{W}_i\}_{i \in [N]} \leftarrow \mathsf{DB}$
2: **for** $i = 1$ to N **do**
3:   $\mathsf{XSet}[i] \leftarrow \emptyset$, $\mathsf{CSet}[i] \leftarrow$ empty multi-map
4:   **for** $w \in \mathsf{W}_i$ and $(\mathsf{ind}, \{attr^*_t, w^*_t\}_{t \in [T]}) \in \mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(w)$ **do**
5:     **if** $\exists j : (\mathbf{w}_1[j] = w \vee \mathbf{w}_2[j] = w) \wedge w^*_t \in \{\mathsf{val}^* : (\mathsf{ind}_1, \mathbf{attr}^*_{\mathbf{t}_1}[j], \mathsf{val}^*) \in \mathsf{Tab}_{\mathbf{t}_1[j]} \wedge (\mathsf{ind}_2, \mathbf{attr}^*_{\mathbf{t}_2}[j], \mathsf{val}^*) \in \mathsf{Tab}_{\mathbf{t}_2[j]} \wedge (\mathsf{ind}_1, \mathbf{w}_1[j]) \in \mathsf{Tab}_{\mathbf{t}_1[j]} \wedge (\mathsf{ind}_2, \mathbf{w}_2[j]) \in \mathsf{Tab}_{\mathbf{t}_2[j]}\}$ **then**
6:       $\mathsf{XSet}[i] \leftarrow \mathsf{XSet}[i] \cup \{H[w, attr^*_t, w^*_t]\}$
7:       $K_{\mathsf{enc},w} \xleftarrow{\$} \{0,1\}^\lambda$
8:       $ct \leftarrow \mathsf{Enc}(K_{\mathsf{enc},w}, 0^\lambda)$
9:       $\mathsf{CSet}[i] \leftarrow \mathsf{CSet}[i].\mathsf{Put}(H[w, attr^*_t, w^*_t], ct)$
10:     **else**
11:       $h \xleftarrow{\$} \{0,1\}^\lambda$
12:       $\mathsf{XSet}[i] \leftarrow \mathsf{XSet}[i] \cup \{h\}$
13:       $K_{\mathsf{enc},w} \xleftarrow{\$} \{0,1\}^\lambda$
14:       $ct \leftarrow \mathsf{Enc}(K_{\mathsf{enc},w}, 0^\lambda)$
15:       $\mathsf{CSet}[i] \leftarrow \mathsf{CSet}[i].\mathsf{Put}(h, ct)$
16:     **end if**
17:   **end for**
18: **end for**
19: **return** $(\mathsf{XSet}, \mathsf{CSet})$

GenTrans$(\mathsf{DB}, \mathsf{EDB}, H, t_1, t_2, w_1, w_2, attr^*_{t_1}, attr^*_{t_2}, \mathsf{stag}, i)$

1: $\mathsf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$
2: $\{(\overline{\mathsf{ind}}_1, \{attr^*_t, w^*_t\}_{t \in [T]}), \ldots, (\overline{\mathsf{ind}}_{T_{w_1}}, \{attr^*_t, w^*_t\}_{t \in [T]})\} \leftarrow \mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_{t_1}}(w_1)$
3: Get $(\bar{w}^*_1, \bar{w}^*_2, \ldots, \bar{w}^*_{T_{w_1}})$ according to $\mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_{t_1}}(w_1)$ and $attr^*_{t_1}$
4: **for** $c = 1, \ldots, T_{w_1}$ **do**
5:   $y \leftarrow \mathsf{t}[c]$
6:   **if** $\bar{w}^*_c \in \{\mathsf{val}^* : (\mathsf{ind}_1, attr^*_{t_1}, \mathsf{val}^*) \in \mathsf{Tab}_{t_1} \wedge (\mathsf{ind}_2, attr^*_{t_2}, \mathsf{val}^*) \in \mathsf{Tab}_2 \wedge (\mathsf{ind}_1, w_1) \in \mathsf{Tab}_{t_1} \wedge (\mathsf{ind}_2, w_2) \in \mathsf{Tab}_{t_2}\}$ **then**
7:     $\mathsf{xjointoken}_1[c] \leftarrow H[w_1, attr^*_{t_1}, \bar{w}^*_c] \oplus y$
8:     $\mathsf{xjointoken}_2[c] \leftarrow H[w_2, attr^*_{t_2}, \bar{w}^*_c] \oplus y$
9:   **else if** $\exists j \neq i : \bar{w}^*_c \in \{\mathsf{val}^* : (\mathsf{ind}, \mathbf{w}_1[j]) \in \mathsf{Tab}_{\mathbf{t}_1[j]} \wedge (\mathsf{ind}, \mathbf{attr}^*_{\mathbf{t}_1}[j], \mathsf{val}^*) \in \mathsf{Tab}_{\mathbf{t}_1[j]}\} \cap \{\mathsf{val}^* : (\mathsf{ind}, w_1) \in \mathsf{Tab}_{t_1} \wedge (\mathsf{ind}, attr^*_{t_1}, \mathsf{val}^*) \in \mathsf{Tab}_{t_1}\}$ **then**
10:     $\mathsf{xjointoken}_1[c] \leftarrow H[w_1, attr^*_{t_1}, \bar{w}^*_c] \oplus y$
11:     $\mathsf{xjointoken}_2[c] \leftarrow H[w_2, attr^*_{t_2}, \bar{w}^*_c] \oplus y$
12:   **end if**
13: **end for**
14: **for** $c = T_{w_1} + 1, \ldots, T_{\max}$ **do**
15:   $\mathsf{xjointoken}_1[c] \xleftarrow{\$} \{0,1\}^\lambda$
16:   $\mathsf{xjointoken}_2[c] \xleftarrow{\$} \{0,1\}^\lambda$
17: **end for**
18: $\mathsf{ResCT} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (t_1, t_2, \mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2))$
19: $q \leftarrow (t_1, t_2, w_1, w_2, attr^*_{t_1}, attr^*_{t_2})$
20: $\mathsf{ResInd} \leftarrow \mathsf{DB}(q)$
21: **return** $((\mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2), \mathsf{ResCT}, \mathsf{ResInd})$

Algorithm 8 Simulator

Initialize($\mathcal{L}_{\mathsf{JXT+}}(\mathsf{DB}, \mathbf{q}), \mathcal{L}_{\mathsf{T}}(\mathsf{DB}, \mathbf{t}_1, \mathbf{w}_1, \mathbf{attr}^*_{\mathbf{t}_1})$)

1: $(\mathsf{n}, \mathsf{RP}, \mathsf{EP}_1, \mathsf{EP}_2, \mathsf{SP}_1, \mathsf{JD}) \leftarrow \mathcal{L}_{\mathsf{JXT+}}(\mathsf{DB}, \mathbf{q})$
2: **for** $w^* \in \bigcup_{i \in [Q]} \mathsf{JD}[i]$ **do**
3: $\quad H[\mathsf{EP}_1[i], w^*] \leftarrow \{0,1\}^\lambda$
4: **end for**
5: **for** $w \in \overline{\mathsf{EP}}_2$ and $w^* \in \bigcup_{i \in [Q]} \mathsf{JD}[i]$ **do**
6: $\quad H[w, w^*] \xleftarrow{\$} \{0,1\}^\lambda$
7: **end for**
8: **for** $i = 1$ to N **do**
9: $\quad \mathsf{T}_i \leftarrow$ empty array
10: $\quad$ **for** $w \in \mathsf{W}_i$ **do**
11: $\quad\quad$ **for** $c = 1, \ldots, T_w$ **do**
12: $\quad\quad\quad$ **for** $t \in [T]$ **do**
13: $\quad\quad\quad\quad y \leftarrow \{0,1\}^\lambda$
14: $\quad\quad\quad\quad \mathsf{T}_i[w||t][c] \leftarrow y$
15: $\quad\quad\quad$ **end for**
16: $\quad\quad$ **end for**
17: $\quad$ **end for**
18: **end for**
19: $(\mathsf{TSet}, \mathsf{STags}) \leftarrow \mathcal{S}_{\mathsf{T}}\big(\mathcal{L}_{\mathsf{T}}(\mathsf{DB}, \mathbf{t}_1, \mathbf{w}_1, \mathbf{attr}^*_{\mathbf{t}_1}), \mathsf{T}_{\mathbf{t}_1}[\mathbf{w}_1||\mathbf{attr}^*_{\mathbf{t}_1}]\big)$
20: **for** $i = 1$ to N **do**
21: $\quad \mathsf{XSet}[i] \leftarrow \emptyset, \mathsf{X}[i] \leftarrow 0, \mathsf{CSet}[i] \leftarrow$ empty multi-map
22: **end for**
23: **for** $w \in \overline{\mathsf{EP}}_2$ and $w^* \in \bigcup_{i \in [Q]:\overline{\mathsf{EP}}_2[i]=w} \overline{\mathsf{RP}}[i]$ **do**
24: $\quad \mathsf{XSet}[\mathbf{t}_2[i]] \leftarrow \mathsf{XSet}[\mathbf{t}_2[i]] \cup \{H[w, w^*]\}$
25: $\quad \mathsf{X}[\mathbf{t}_2[i]] \leftarrow \mathsf{X}[\mathbf{t}_2[i]] + 1$
26: $\quad K_{\mathsf{enc},w} \xleftarrow{\$} \{0,1\}^\lambda$
27: $\quad ct \leftarrow \mathsf{Enc}(K_{\mathsf{enc},w}, 0^\lambda)$
28: $\quad \mathsf{CSet}[\mathbf{t}_2[i]] \leftarrow \mathsf{CSet}[\mathbf{t}_2[i]].\mathsf{Put}(H[w, w^*], ct)$
29: **end for**
30: **for** $w \in \mathsf{EP}_1$ and $w^* \in \bigcup_{i \in [Q]:\mathsf{EP}_1[i]=w} \overline{\mathsf{RP}}[i]$ **do**
31: $\quad \mathsf{XSet}[\mathbf{t}_1[i]] \leftarrow \mathsf{XSet}[\mathbf{t}_1[i]] \cup \{H[w, w^*]\}$
32: $\quad \mathsf{X}[\mathbf{t}_1[i]] \leftarrow \mathsf{X}[\mathbf{t}_1[i]] + 1$
33: $\quad K_{\mathsf{enc},w} \xleftarrow{\$} \{0,1\}^\lambda$
34: $\quad ct \leftarrow \mathsf{Enc}(K_{\mathsf{enc},w}, 0^\lambda)$
35: $\quad \mathsf{CSet}[\mathbf{t}_1[i]] \leftarrow \mathsf{CSet}[\mathbf{t}_1[i]].\mathsf{Put}(H[w, w^*], ct)$
36: **end for**
37: **for** $i = 1$ to N **do**
38: $\quad$ **for** $j = \mathsf{X}[i] + 1, \ldots, \mathsf{n}[i] \cdot T$ **do**
39: $\quad\quad h \xleftarrow{\$} \{0,1\}^\lambda$
40: $\quad\quad \mathsf{XSet}[i] \leftarrow \mathsf{XSet}[i] \cup \{h\}$
41: $\quad\quad K_{\mathsf{enc},w} \xleftarrow{\$} \{0,1\}^\lambda$
42: $\quad\quad ct \leftarrow \mathsf{Enc}(K_{\mathsf{enc},w}, 0^\lambda)$
43: $\quad\quad \mathsf{CSet}[i] \leftarrow \mathsf{CSet}[i].\mathsf{Put}(h, ct)$
44: $\quad$ **end for**
45: **end for**
46: $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet}, \mathsf{CSet})$
47: **for** $i = 1$ to Q **do**
48: $\quad \mathbf{Tr}[i] \leftarrow \mathbf{GenTrans}(\mathsf{EDB}, H, \mathbf{t}_1[i], \mathbf{t}_2[i], \mathbf{w}_1[i], \mathbf{w}_2[i], \mathbf{attr}^*_{\mathbf{t}_1}[i], \mathbf{attr}^*_{\mathbf{t}_2}[i], \mathsf{STags}[i], i, \mathsf{RP}, \mathsf{SP}_1, \mathsf{JD})$
49: **end for**
50: **return** $(\mathsf{EDB}, \mathbf{Tr})$

GenTrans($\mathsf{EDB}, H, t_1, t_2, w_1, w_2, attr^*_{t_1}, attr^*_{t_2}, \mathsf{stag}, i, \mathsf{RP}, \mathsf{SP}_1, \mathsf{JD}$)

1: $\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$
2: $R \leftarrow \overline{\mathsf{RP}}[i] \cup \bigcup_{j=1}^{Q} \mathsf{JD}[j] \cap \mathsf{JD}[i]$
3: $(\bar{w}^*_1, \bar{w}^*_2, \ldots, \bar{w}^*_{|R|}) \leftarrow R$, where $|R| \leq \mathsf{SP}[i]$
4: $\bar{w}^*_k \leftarrow \perp$ for $k = |R| + 1, \ldots, \mathsf{SP}[i]$
5: **for** $c = 1, \ldots, \mathsf{SP}[i]$ **do**
6: $\quad y \leftarrow \mathbf{t}[c]$
7: $\quad$ **if** $\bar{w}^*_c \neq \perp$ **then**
8: $\quad\quad \mathsf{xjointoken}_1[c] \leftarrow H[w_1||attr^*_{t_1}, \bar{w}^*_c] \oplus y$
9: $\quad\quad \mathsf{xjointoken}_2[c] \leftarrow H[w_2||attr^*_{t_2}, \bar{w}^*_c] \oplus y$
10: $\quad$ **else**
11: $\quad\quad \mathsf{xjointoken}_1[c] \xleftarrow{\$} \{0,1\}^\lambda$
12: $\quad\quad \mathsf{xjointoken}_2[c] \xleftarrow{\$} \{0,1\}^\lambda$
13: $\quad$ **end if**
14: **end for**
15: **for** $c = \mathsf{SP}[i] + 1, \ldots, T_{\max}$ **do**
16: $\quad \mathsf{xjointoken}_1[c] \xleftarrow{\$} \{0,1\}^\lambda$
17: $\quad \mathsf{xjointoken}_2[c] \xleftarrow{\$} \{0,1\}^\lambda$
18: **end for**
19: $\mathsf{ResCT} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (t_1, t_2, \mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2))$
20: $\mathsf{ResInd} \leftarrow \mathsf{RP}[i]$
21: **return** $((\mathsf{stag}, \mathsf{xjointoken}_1, \mathsf{xjointoken}_2), \mathsf{ResCT}, \mathsf{ResInd})$