Efficient Searchable Symmetric Encryption for Join Queries

Charanjit Jutla[1], Sikhar Patranabis[2]

[1] IBM Research, New York, USA
csjutla@us.ibm.com
[2] IBM Research, Bangalore, India**
sikhar.patranabis@ibm.com

Abstract. The Oblivious Cross-Tags (OXT) protocol due to Cash et al. (CRYPTO'13) is a highly scalable searchable symmetric encryption (SSE) scheme that allows fast processing of conjunctive and more general Boolean queries over encrypted relational databases. A longstanding open question has been to extend OXT to also support queries over joins of tables without pre-computing the joins. In this paper, we solve this open question without compromising on the nice properties of OXT with respect to both security and efficiency. We propose Join Cross-Tags (JXT) - a purely symmetric-key solution that supports efficient conjunctive queries over (equi-) joins of encrypted tables without any pre-computation at setup. The JXT scheme is fully compatible with OXT, and can be used in conjunction with OXT to support a wide class of SQL queries directly over encrypted relational databases. JXT incurs a storage cost (over OXT) of a factor equal to the number of potential join-attributes in a table, which is usually compensated by the fact that JXT is a fully symmetric-key solution (as opposed to OXT which relies on discrete-log hard groups). We prove the (adaptive) simulation-based security of JXT with respect to a rigorously defined leakage profile.

1 Introduction

The advent of cloud computing allows individuals and organizations to outsource the storage and processing of large volumes of data to third party servers. However, clients typically do not trust service providers to respect the confidentiality of their data [CZH$^+$13]. This lack of trust is often further reinforced by threats from malicious insiders and external attackers. One solution is to upload data in an encrypted form, with the client keeping the secret key.

Consider a client that offloads an encrypted relational database of (potentially sensitive) credit-card transactions to an untrusted server. At a later point of time, the client might want to issue a query of the form *retrieve all transactions for a particular merchantID for a given time*. Ideally, we want the client to be able to perform this task without revealing any sensitive information to

** Corresponding author. Most of the work was done while the author was affiliated with ETH Zürich, Switzerland and Visa Research USA.

the server, such as the actual transactions, the merchantID underlying the given query, etc. Moreover, one could consider even more complicated Boolean queries over additional *attribute-value pairs*. Unfortunately, techniques such as fully homomorphic encryption [Gen09], that potentially allow achieving such an "ideal" notion of privacy, are unsuitable for practical deployment due to large performance overheads.

Searchable Symmetric Encryption. Searchable symmetric encryption (SSE) [SWP00,Goh03,CGKO06,CK10,PRZB11a,CJJ$^+$13,CJJ$^+$14,KM18,CNR21] is the study of provisioning symmetric-key encryption schemes with search capabilities. The most general notion of SSE with optimal security guarantees can be achieved using the work of Ostrovsky and Goldreich on Oblivious RAMs [GO96]. More precisely, using these techniques, one can evaluate a functionally rich class of queries on encrypted data without leaking *any* information to the server. However, such an ideal notion of privacy comes at the cost of significant computational or communication overhead.

A large number of existing SSE schemes prefer to trade-off security for practical efficiency by allowing the server to learn "some" information during query execution. The information learnt by the server is referred to as *leakage*. Some examples of leakage include the database size, the query pattern (which queries have the same attribute-value pair) and the access pattern. Practical implementations of such schemes can be made surprisingly efficient and scalable using specially designed data structures. This line of works on efficient SSE schemes that trade-off leakage for efficiency was initiated by Curtmola et al. [CGKO06], who introduced and formalized the simulation-based framework for proving the security of SSE schemes with respect to a given leakage function. Subsequently, Chase and Kamara [CK10] introduced the concept of "structured encryption" - a generalization of SSE to structured databases, along with the corresponding security definitions.

For any SSE scheme to be truly practical, it should at least support conjunctive queries, i.e., given a set of attribute-value pairs $(\mathsf{w}_1, \ldots, \mathsf{w}_n)$, it should be able to find and return the set of records that match *all* of these attribute-value pairs. The example query above, namely, "*retrieve all transactions for a particular merchantID for a given time*" is an instance of a conjunctive query. There exist dedicated SSE schemes that can support conjunctive, disjunctive and general Boolean queries over attribute-value pairs in relational databases [CJJ$^+$13,CJJ$^+$14,KM17,LPS$^+$18,PM21].

A very important class of queries that any relational database should support are queries over joins of tables. We illustrate the concept of joins by extending the above example. Consider the scenario where the credit-card processor has two tables: (Table A) a transactions table and (Table B) a merchants information table. Instead of the earlier query "*retrieve all transactions for a particular merchantID for a given time*" in Table A, the new query may be "*retrieve all records for a given time* HHMM *in a given city* CC" in the *join* of Table A and Table B, where the join is over the attribute merchantID. More formally, the result of such a query is

$$\begin{aligned}&\{(\langle \mathsf{A};\, \mathsf{r1}\rangle, \langle \mathsf{B};\, \mathsf{r2}\rangle) \mid \exists\, \text{merchantID } \textsc{mid} \;:\\ &\quad (\text{recordID}= \langle \mathsf{A};\, \mathsf{r1}\rangle, \text{time}= \textsc{hhmm}, \text{merchantID} = \textsc{mid}) \in \text{Table A}\\ &\qquad \textbf{and}\\ &\quad (\text{recordID}= \langle \mathsf{B};\, \mathsf{r2}\rangle, \text{city}= \textsc{cc}, \text{merchantID} = \textsc{mid}) \in \text{Table B}\}\end{aligned}$$

Unfortunately, the above schemes are unable to handle such queries on joins of tables without prohibitive pre-computation of joins. This inability to efficiently and flexibly support queries over joins of tables is indeed a major impediment to actual deployment of these schemes. Only a handful of recent works [KM18,CNR21] address search queries over joins of tables; we will review their techniques subsequently.

Oblivious Cross-Tags (OXT). The work of Cash et al. [CJJ$^+$13] showed for the first time how to design an SSE scheme for conjunctive (and more general Boolean) queries, for which (i) the encrypted database has memory requirement that is linear in the size of the database, (ii) searches require a single round of communication (query followed by response), and (iii) the leakage to the server is low. Their scheme, called Oblivious Cross-Tags (OXT), relies on specially *structured* pseudorandom functions (PRFs), such as those that can be enabled using hard discrete-log groups.

Since our work is closely related to OXT, we give a brief overview. In its simplest embodiment, the SSE scheme OXT precomputes an encrypted version of a database (using a secret symmetric key) and stores it at a server that is presumed to be *honest-but-curious.* A client with access to this symmetric key, breaks a (2-) conjunctive query $\mathfrak{b} = \mathfrak{b}_1 \wedge \mathfrak{b}_2$ into two search tokens for the server. The first search token yields all entries for the first conjunct $\mathfrak{b}_1$ and the second search token is used to search for exactly the conjunct $\mathfrak{b}_2$ using a "*cross-tag helper token*" stored as part of the entries for $\mathfrak{b}_1$. The cross-tag helper is independent of the second attribute and hence only one cross-tag helper per record-attribute pair is stored. Since there is one data element anyway for each record-attribute pair, this at most doubles the total space requirement. For example, consider the conjunctive query above: (time HHMM and merchantID MID). The client computes two PRF values (using its secret key): one for (time; HHMM), say $\mathsf{p1}$, and another for (merchantID; MID), say $\mathsf{p2}$. It sends to the server a key $\mathsf{k1}$ derived from $\mathsf{p1}$, and a token $=\ \mathsf{h}^{\mathsf{p2}/\mathsf{p1}}$ (in a DDH-hard group with generator h). The attributes (time, merchantID) are also revealed to the server.

The server uses $\mathsf{k1}$ to search for an encrypted set (stored in the encrypted database) corresponding to (time; HHMM) as well as uses $\mathsf{k1}$ to decrypt it. Next, for each record, in this decrypted set D, the server can also find a "*cross-tag helper token*" $\mathsf{z} = \mathsf{p1} * \mathsf{rind}$ (where, rind stands for randomized-record-index). The search token $\mathsf{h}^{\mathsf{p2}/\mathsf{p1}}$ raised to the power z yields a *cross-tag* $\mathsf{h}^{\mathsf{p2}*\mathsf{rind}}$, which is then checked in a lookup-table called XSet. This lookup-table XSet stored with the server has every valid member of the form $\mathsf{h}^{\mathsf{p2}*\mathsf{rind}}$, and hence this check allows the server to confirm the record in D to satisfy the conjunct. Note, the size of this set XSet is exactly the number of records times the number of attributes (as in each record, for each attribute there is exactly one value such as MID). This is exactly the size of the database. That this lookup-table reveals no information, a priori, is proved under the DDH assumption[3]; hence the name *oblivious cross-tag* (OXT).

Note that, both the client and the server have to perform exponentiations (in the DDH-hard group) during this search protocol. Moreover, the number of these exponentiations can be large, as there will be one such exponentiation per entry in the decrypted set D. Similarly, during the setup stage, i.e. when the database is encrypted and the XSet is computed, an exponentiation is required for every attribute-value pair in the database. Hence, the setup maybe computationally intensive for large databases.

1.1 The New JXT Protocol

Our first contribution is to show that if the number of attributes in a table is small, say m, then the encrypted database with a size blowup by a factor m, can achieve the same security as OXT without the use of DDH, and more precisely using only symmetric-key primitives such as PRFs and symmetric-key encryption in the standard model. As a result, the search computation becomes considerably faster as there is no exponentiation (by either the client or the server). Further, the setup becomes much faster, as the XSet computation requires no exponentiations.

Next, as a main contribution of this work, we show that the above modification to OXT also allows us to search over (equi-) joins of tables without any pre-computation of joins[4]. We refer to this new protocol as Join Cross-Tags (abbreviated as JXT). Moreover, since joins are usually performed over a limited set of attributes (e.g. primary keys or high-entropy attributes[5]), the size blowup to the encrypted database is small; more precisely, a T fold blowup, where T is the number of attributes in a table over which joins are allowed. Recall the example join query "*retrieve all records for a given time* HHMM *in a given city* CC" in the join of Table A and Table B, where the join is over the attribute merchantID. The JXT protocol can support this query (over encrypted databases) without any pre-computation of joins of the two tables. The only requirement is that both encrypted tables must be configured to support join over the attribute merchantID[6]. As mentioned previously, if merchantID is amongst the few attributes (say, T many) that a table supports for join, the space requirement for that encrypted

[3] The actual protocol is slightly more complicated to be fully secure and provably secure under DDH, but the above description gives the main gist of OXT.

[4] Throughout this paper, when we refer to joins, we mean equi joins.

[5] By high entropy attribute we mean the information-theoretic entropy of the column corresponding to the attribute. For example, the attribute *gender* has low entropy, whereas the attribute *name* can have high entropy in a table.

[6] By configuration we mean the (pre-) computation of the encrypted table. We remind the reader that this pre-computation does not involve join pre-computation, as each table is encrypted independently.

table only increases T-fold. Some other tables may not even have the attribute merchantID, but these may have other small number of attributes over which join is allowed.

We provide a more detailed overview of the ideas behind the JXT protocol subsequently in Section 1.2.

Comparison with Pre-Computation of Joins. It is worth contrasting this approach to one where joins are pre-computed (for instance, in [KM18,CNR21] as discussed under related work in Section 1.4 later), and this is best exhibited by considering the above example. The transactions table A is likely to be tall and skinny, i.e. have many records and few attributes. On the other hand, the merchant information table B is likely to be short and wide. However, their join will have at least as many records as table A and at least as many attributes as table B, i.e. tall and wide. This can cause a considerable blowup in storage requirement. Since JXT does not pre-compute joins, it avoids such blowups, as well as other blowups caused by the possibilities of pair-wise joins of many tables. We present a more detailed comparison of JXT with the above pre-processing based approaches to Section 1.4.

Modular Setup and Flexible Updates. The JXT approach also allows for a modular setup stage, as well as flexible table additions and updates. Some tables are updated much faster than others, and hence can be re-setup on their own without the need to re-setup tables that are more or less constant[7]. This allows flexibly adding new tables and updating existing tables in an independent manner without having to re-perform setup across all tables in the encrypted database. This is not supported by any of the existing approaches where joins are pre-computed [KM18,CNR21], and constitutes one of the most appealing aspects of our JXT construction.

Storage and Search Overheads. We provide a high-level summary of the overheads incurred by JXT in terms of storage and search processing. Suppose that a table has a total of n attributes, with $\text{T} \leq n$ amongst these being "join attributes"; i.e. attributes over which the table can be joined with other tables in the database. Also, assume that the table has a total of m records (equivalently, rows). Then, in JXT, the corresponding encrypted table incurs a storage overhead of $O(mn\text{T})$, which is $O(\text{T})$-fold blowup to the storage required for the plaintext table. Also, given a 2-conjunctive query over the join of two tables that involves an attribute-value pair w_1 from the first table and an attribute-value pair w_2 from the second table, the computational overhead at the server is $O(\ell_1\ell_2)$, where ℓ_1 and ℓ_2 are the numbers of records matching the attribute-value pairs w_1 and w_2 in the first and second table, respectively.

[7] We remark here that the transactions database is encrypted for post-transactional audit, fraud detection, money-laundering detection, machine learning etc. The real-time transactions database is usually updated and used without encryption, as it runs in a secure domain. It is later encrypted on a periodic basis for above additional functionalities.

Compatibility with OXT. An important feature of JXT is that it is fully compatible with OXT. For example, consider the two tables A and B above and suppose table A has few attributes (say e.g. four) and table B has many attributes (say, e.g. twenty). Also, suppose that some of these attributes are the attributes over which joins can be performed. Then, the OXT protocol can be used to support Boolean queries within each table (spanning many attributes), as well as Boolean queries across tables using the JXT part for the join. So for example, the query maybe a 4-conjunct "*retrieve all records for a given time and a given amount in a given city and a given merchant category*" in the join of Table A and Table B, where the join is over the attribute merchantID.

Further, there is a "multi-client" extension of OXT where the client does not own the secret key; instead, an authority owns the secret key and the client computes its PRF based search tokens using an oblivious-PRF (OPRF) protocol with the authority [JJK+13]. JXT is also fully compatible with this multi-client extension of OXT. In fact, JXT can be easily extended to the scenario where different databases are owned by different entities operating under a single authority, and a client can perform a search query over join of tables owned by different entities; this requires that the entities setup their respective encrypted databases using "oblivious" help from the authority.

1.2 The Main Idea of Our JXT Protocol

We now present a brief overview of the main ideas behind our new JXT protocol.

Breaking a Join Query into Sub-Queries. To begin with, we show that a 2-conjunctive (join) query $\mathfrak{q} = \mathfrak{q}_1 \wedge \mathfrak{q}_2$ over the join of a pair of tables (say Tables A and B), with the join being over a third attribute, e.g. merchantID, can be broken into normal sub-queries, i.e. non-join queries, that are either over table A or table B. These sub-queries might be simple or conjunctive, but over a single table. We illustrate this with our running example from above, i.e. the query "*retrieve all records for a given time* HHMM *in a given city* CC" in the *join* of Table A and Table B, where the join is over the attribute merchantID. In this case, the *first sub-query* $\mathfrak{a}$ can be viewed as the simple keyword search for the attribute-value pair (time= HHMM) in Table A. Now, suppose that for all records matching this first sub-query $\mathfrak{a}$ in Table A, we create a set of the corresponding values of merchantID, say of the form {MID}. We can now define a *second set of sub-queries* $\mathfrak{B}$ in Table B, with one sub-query for each MID in the aforementioned set. Each such sub-query $\mathfrak{b}$ (in the set $\mathfrak{B}$) is of the form "*retrieve all records for a given city* CC *for a given merchantID* MID". Note that $\mathfrak{b}$ is a 2-conjunctive query itself.

Handling Sub-Queries. It is easy to see that each sub-query $\mathfrak{b}$ in $\mathfrak{B}$ can be executed securely using the original OXT protocol, if we could somehow use the results of the first sub-query $\mathfrak{a}$ to derive the tokens needed by OXT server to execute $\mathfrak{b}$. In other words, we wish to design a protocol such that executing $\mathfrak{a}$ in Table A generates a set of tokens that an OXT server can use in the same way as it would use tokens issued directly by an OXT client for query $\mathfrak{b}$ in Table B. The challenging part is to implement the search in Table A to mimic this client for OXT query $\mathfrak{b}$ in Table B, without introducing additional rounds of interaction between the client and the server. Achieving this constitutes the technical core of our new JXT protocol.

As explained earlier in the Introduction, in the original OXT protocol, a client with access to the symmetric key, breaks a 2-conjunctive query $\mathfrak{b} = \mathfrak{b}_1 \wedge \mathfrak{b}_2$ into two search tokens for the OXT server. Similarly, to enable the JXT protocol, we would need two search tokens for each $\mathfrak{b}$ in $\mathfrak{B}$, as it is a 2-conjunctive query. The first search token should yield all entries for the first conjunct $\mathfrak{b}_1$ ((city=CC) for our example), and the second search token is used to search for the second conjunct $\mathfrak{b}_2$ ((merchantID=MID), in our example) using a '*cross-tag helper token*" stored (in the OXT server) as part of the entries for the first conjunct. The OXT client computes two PRF values (using its secret key): one for $\mathfrak{b}_1$ (city; CC), say $\mathsf{p1}$, and another for $\mathfrak{b}_2$ (merchantID; MID), say $\mathsf{p2}$. It sends to the server a key $\mathsf{k1}$ derived from $\mathsf{p1}$, and a token = $\mathsf{h}^{\mathsf{p2}/\mathsf{p1}}$ (in a DDH-hard group with generator h). In the JXT protocol, we require that executing $\mathfrak{a}$ in Table A precisely generates this token $\mathsf{h}^{\mathsf{p2}/\mathsf{p1}}$.

Implementing Search in Table A. The first challenge we face is to implement the search in Table A in a manner that mimics the client for the second set of OXT queries in Table B. We achieve this as follows. At a high level, our idea is to amend the encrypted table A to store the PRF value $\mathsf{p2}$ for the merchantID MID in each record in the set keyed by (time; HHMM), such that the search token in query $\mathfrak{a}$ (corresponding to (time= HHMM)) can be used to retrieve these $\mathsf{p2}$ values. However, note that doing this naïvely has two disadvantages. First of all, it would reveal the occurrence of the same $\mathsf{p2}$ value across many different queries. More crucially, this potentially causes a quadratic blowup in storage, since we would need to store the $\mathsf{p2}$ value for each attribute-value pair in the record, when storing it as a set of records keyed by say, (time, HHMM).

We tackle this as follows. If we restrict the join attributes to be a limited set, say of size T, then the blowup is only T-fold. This is a reasonable assumption in practice, since the join attribute is typically either the primary key (i.e. takes a unique value for each record) or a high-entropy attribute, and there are likely to be only a limited number of candidate join attributes per table. In order to hide the occurrences of join attributes across different queries, we embed nonces or counters in the pseudorandom values. As in OXT, this allows us to avoid cross-query leakage.

Implementing Search in Table B. Since we are willing to allow a T-fold blowup in the encrypted database, we now show that with a 2*T-fold blowup, we can actually get rid of the complicated discrete-log based approach of OXT for handling conjunctive queries, at least for the OXT part that we are emulating inside JXT. Recall in OXT, for each first conjunct $\mathfrak{b}_1$ we stored the "cross-tag helper token" $\mathsf{z} = \mathsf{p}_1 * \mathsf{rind}$. Instead, we now store T different helper tokens $\mathsf{z}_t = \mathsf{p}_1 * \mathsf{rind}_t$ (or simply $\mathsf{p}_1 + \mathsf{rind}_t$), where rind_t is a different pseudorandom

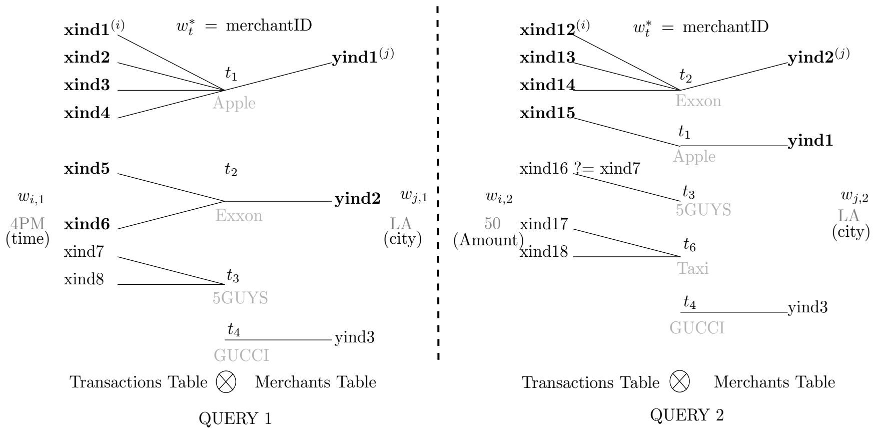

Fig. 1: We illustrate here how the leakage across join queries depends on the ordering of the attribute-value pairs and/or the join ordering. Query 1 is "SELECT * FROM (Transactions JOIN Merchants ON merchantID) WHERE time = 4PM AND city = LA." Query 2 is "SELECT * FROM (Transactions JOIN Merchants ON merchantID) WHERE amount = 50 AND city = LA." The *conditional intersection pattern leakage* reveals that yind1 is same in the two queries. Even though xind16 maybe same as xind7, this information is not leaked. The *join-distribution pattern leakage* also reveals that in the Transactions table, there are additional sets of records, each record in the set sharing the same merchantID (i.e., (redacted) 5GUYS from Query 1 has two records and (redacted) Taxi from Query 2 has two records as well). However, if the order in which the tables are joined is reversed, i.e., the query is over "Merchants JOIN Transactions", then the *join-distribution leakage* is null. This is because merchantID is the *primary key* in the Merchants table. For more details, see Section 5.1.

value for each $t \in [\mathrm{T}]$. This way, the search token for $\mathfrak{b}_2$ need not be $\mathsf{h}^{\mathsf{p2}/\mathsf{p1}}$ anymore, but just $\mathsf{p2} - \mathsf{p1}$. This, when added to the particular cross-tag helper token would yield rind_t, which can then be checked for membership[8] in XSet_t. Security is maintained because for different t, rind_t is random and independent. The search token for $\mathfrak{b}_2$ is now just $\mathsf{p2} - \mathsf{p1}$, which is readily obtained from the search for $\mathfrak{b}_1$ in table A as described above. Of course, the actual protocol is slightly more complicated as we embed nonces in the PRF values to attain full security. The detailed protocol for JXT appears in Section 3 and Figures 2 and 3.

1.3 The Leakage Profile of JXT

An astute reader may wonder about the leakage of JXT and how it compares to the leakage profile of the OXT protocol. The leakage profile of OXT (i.e. leakage to the server) is known to be technically abstruse, but at the same time a careful analysis also shows that in practice the leakage of OXT is benign given that much

[8] Note that we have T different XSets, but their total size is same as the single XSet of OXT.

of this leakage can also be obtained a priori by auxiliary means. We remark that the OXT leakage profile is abstruse mainly because the OXT protocol achieves high scalability while supporting Boolean search queries. Further, the rigorous definition of the leakage profile allows for a simulation-based security proof of OXT. The leakage profile for JXT also follows the same model, with some additional leakage over OXT leakage, which is to be expected because the queries are across tables and express an existential quantifier over the join attribute. Nevertheless, we describe below that in practice the leakage is still benign. In this Introduction section, this is best illustrated using an example as in Figure 1. A rigorous definition of the leakage profile is given in Section 5.1.

The leakage of JXT can be split into six main categories: (a) database size, (b) result pattern of the queries, (c) equality pattern across the queries, (d) size pattern of the queries, (e) conditional intersection pattern across queries, and (f) join-attribute distribution pattern of the queries. While the first five are more or less similar to OXT leakage (but, see Section 5.1 for subtle differences), the last one is obviously new to JXT.

We illustrate the concept of join-attribute distribution pattern leakage using the running example. For the query over the join of Tables A and B with respect to the attribute merchantID, it reveals the frequency distribution of values taken by merchantID across records matching the attribute-value pair in the "first slot", i.e. records in Table A matching the time HHMM.

The extent of this leakage depends on the "entropy" of the join attribute merchantID in the first table i.e. Table A. In this particular example, merchantID is the primary key in Table A, and takes a unique value for each record. Hence, in this case, the join-attribute distribution pattern leakage is essentially query-invariant (as each possible value occurs with frequency exactly 1), and hence, benign. In other examples, the join attribute may not be a primary key in the first table. However, if it is still a "high-entropy" attribute in the first table, it is likely to take a unique value for each record (or each value with frequency close to 1), and hence, the leakage can be minimal.

Now, just as in OXT, the client has the choice to order the conjuncts in a query, as well as the order in which the tables are joined. The way OXT is designed is that the first conjunct usually leaks the most information (as the server gets to decrypt information related to the the first conjunct). Thus, usually, the attribute that has lesser entropy is not made the first conjunct in a query, as the size pattern leakage has the potential to un-blind the attribute-value pair for a low-entropy attribute (such as gender). A similar design principle is followed in JXT, and the order of the tables being joined can make a difference to the leakage, as illustrated in Figure 1. In particular, the table in which the join attribute is the primary-key (or high-entropy attribute) should be made the first table in a join query. If such an ordering is always possible, then the additional join-attribute distribution pattern leakage can be null (in case of primary-keys) or minimal (in case of high-entropy attributes).

Security Proofs. We formally prove the (adaptive) simulation-based security of JXT with respect to the above leakage profile (formally defined in Section 5.1).

Our proofs follow the same simulation-based framework that was originally proposed by Curtmola et al. in [CGKO06] (and is widely adopted in the SSE literature [CK10,CJJ+13]). Our proofs employ purely symmetric-key primitives such as PRFs and symmetric-key encryption in the standard model.

1.4 Related Work

The SPX Scheme. In [KM18], Kamara and Moataz showed how to encrypt a relational database in such a way that it can efficiently support a large class of SQL queries, including join queries. However, their proposed protocol (called SPX) crucially relies on explicitly pre-computing joins over all attributes that share a common domain. In our context, SPX essentially pre-computes joins of tables over all attributes configured for joins. On the other hand, our proposed JXT protocol avoids all such pre-computation of joins (and the associated storage overheads as discussed earlier).

We note, however, that there are scenarios where SPX provides better query complexity than JXT. Consider a join query over tables Tab_1 and Tab_2, where each individual table has m_1 and m_2 matching records, respectively, but the number of matching records in the join of Tab_1 and Tab_2 is *empty*. This is an extreme case where SPX outperforms JXT since the computational and communication complexity incurred by JXT is $O(m_1 \cdot m_2)$, while that incurred by SPX is $O(1)$.

We also note that the SPX protocol leaks less information during join queries as compared to JXT. This is another consequence of the pre-processing of joins at setup in SPX. In particular, SPX does not incur two kinds of leakage that JXT does: the conditional intersection pattern leakage and the join attribute distribution pattern leakage. We view these types of leakage as tradeoffs for efficiency and flexibility (w.r.t. table updates) that JXT achieves by avoiding pre-computation of joins at setup.

The CNR Scheme. In a more recent work [CNR21], Cash et al. introduced the interesting concept of *partially pre-computed joins*, which potentially has a lower result pattern leakage than is usually expected (we refer to their construction as CNR henceforth). In CNR, the server only learns the projection of the actual result set onto the two tables, and the client has to do extra work to extract the exact set of records matching the join query. However, the storage requirement for their scheme is at least as much as would be required in a scheme that pre-computes joins of tables at setup. Finally, it is not immediately obvious if their scheme is compatible with OXT, which is the state-of-the-art for conjunctive (and more general Boolean) queries. As in SPX, there are scenarios where CNR provides better query complexity than JXT by virtue of the partial pre-computation of joins at setup. Additionally, CNR also does not incur the conditional intersection pattern leakage and the join attribute distribution pattern leakage.

Property-Preserving Encryption. Finally, there exist solutions based on property-preserving encryption (PPE) that allow handling a large class of SQL queries over encrypted relational databases. However, these schemes are vulnerable to *leakage-abuse attacks* [IKK12,NKW15,CGPR15,ZKP16,BKM20]. For example, PPE-based schemes such as CryptDB [PRZB11b] typically use deterministic encryption and its variants to support conjunctive (and other Boolean queries), as well as join queries. These techniques typically leak *frequency information* about the underlying plaintext data, which can be potentially exploited in certain settings to completely break query privacy [NKW15]. Our proposed JXT protocol, on the other hand, does not use any PPE-like techniques, and only incurs benign leakage (similar to those in OXT) that are resistant to leakage-abuse attacks (see [BKM20] for an overview of why leakage incurred by schemes such as OXT are not exploitable via leakage-abuse attacks in practice).

1.5 Open Questions and Future Research Directions

Our work gives rise to many interesting open questions and directions of future research. We summarize some of them below.

Joins over Arbitrary and Low-Entropy Attributes. While it is true that JXT does not support joins over arbitrary attributes (in particular, the attributes over which the encrypted database was not configured to support joins), in practice, it is indeed the case that the designer of the tables knows in advance which attributes are likely candidates for joins. We leave it as an open problem to analyze the leakage of the JXT protocol when a join is performed over an attribute which has "low-entropy" in *both* the tables.

Joins over Three or More Tables. We also leave it as an interesting direction of future research to extend JXT to support queries over joins of *three or more* tables (without pre-computation). We do believe that our techniques presented in this paper can be extended to support joins over three (or more) tables. However, a detailed discussion of such an extension is beyond the scope of this paper, as formalizing the implications for its leakage is likely to be non-trivial and could involve some unexpected issues. In addition, a detailed security proof for such an extension would require careful analysis.

Concretely, we expect the key non-triviality to arise in the search protocol, where the client needs to send to the server a significantly more complicated combination of join tokens to enable searching over joins of three (or more) tables, while leaking as little information as possible beyond the leakage for the two-join case. For instance, a naive extension from two-joins to three-joins might leak whether a particular record is in the join of two tables, but not in the join of all three tables. We would ideally want to avoid such "sub-query leakage", which could otherwise lead to attacks.

JXT for Dynamic Databases. We also leave it open to extend JXT to support dynamic addition/deletion of records directly to the encrypted database (e.g., in the spirit of [PM21], which extends OXT to the dynamic setting). Another open problem is to extend JXT to achieve lower result pattern leakage, as in the scheme due to Cash et al. [CNR21] discussed above.

Implementation. We acknowledge that implementing and testing JXT over massive relational databases with TBs of data is important from a performance analysis point of view. However, given the potentially significant implementation-level challenges involved, we leave this as an interesting and challenging follow-up project (similar to the follow-up to OXT by Cash et al. [CJJ$^+$14]).

2 Definitions and Tools

Notation. We write $[n]$ for the set $\{1, \ldots, n\}$. For a vector $\mathbf{v}$ we write $|\mathbf{v}|$ for the dimension (length) of $\mathbf{v}$ and for $i \in [|\mathbf{v}|]$ we write $\mathbf{v}[i]$ for the i-th component of $\mathbf{v}$. All algorithms (including adversaries) are assumed to be probabilistic polynomial-time (PPT) unless otherwise specified. If A is an algorithm, then $y \leftarrow A(x)$ means that the y is the output of A when run on input x. If A is randomized then y is a random variable. We refer to $\lambda \in \mathbb{N}$ as the security parameter, and denote by poly(λ) and neg(λ) any generic (unspecified) polynomial function and negligible function in λ, respectively. [9]

2.1 Relational Databases and Join Queries

Syntax. A relational database $\mathsf{DB} = \{\mathsf{Tab}_i\}_{i \in [N]}$ is represented as a collection of tables. Each table Tab_i is in turn composed of records over a set of attribute-value pairs W_i. For simplicity, we represent Tab_i as a list of tuples of the form $\{(\mathsf{ind}_{i,\ell}, \mathsf{w}_{i,\ell})\}_{\ell \in [L]}$, where each record-identifier $\mathsf{ind}_{i,\ell}$ is a bit-string in $\{0,1\}^\lambda$ and each attribute-value-pair $\mathsf{w}_{i,\ell} \in \mathsf{W}_i$ is an (arbitrary-length) bit-string in $\{0,1\}^*$. For the sake of search it is sufficient to represent a record as its associated attribute-value pair set W_i.

Identifiers. An identifier $\mathsf{ind}_{i,\ell}$ is a value that can be revealed to the server storing the database (for instance, a permutation of the original record indices). It can be used by the server to efficiently retrieve the corresponding (encrypted) record and send it to the client. We assume throughout the paper that any identifier ind corresponding to a record in a table Tab_i is appended with the table number i. In other words, two distinct tables Tab_i and Tab_j cannot have a record identifier ind in common.

Join Attributes. We assume that for each table T_i, the set of all attributes $\{\mathsf{attr}^*_{i,t}\}_{t \in [T]}$ that it shares with other tables in the database DB is fixed at setup and has size upper-bounded by some polynomial function of the security parameter. We refer to such attributes as "join attributes". Looking ahead, these join attributes are used to perform join queries across tables.

[9] Note that a function $f : \mathbb{N} \to \mathbb{N}$ is said to be negligible in λ if for every positive polynomial p, $f(\lambda) < 1/p(\lambda)$ when λ is sufficiently large.

Inverted Index. For an attribute-value pair $\mathsf{w} \in \mathsf{W}_i$, we define $\mathsf{DB}_{\mathsf{Tab}_i}(\mathsf{w})$ as the set of identifiers of records that contain an entry matching w. In other words, $\mathsf{DB}_{\mathsf{Tab}_i}(\mathsf{w})$ is a set of the form:

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

We refer to the collection of sets $\{\mathsf{DB}_{\mathsf{Tab}_i}(\mathsf{w}_\ell)\}_{\mathsf{w}_\ell \in \mathsf{W}_i}$ as the "inverted index" for the table Tab_i.

Inverted Join Index. For an attribute-value pair $\mathsf{w} \in \mathsf{W}_i$, we additionally define $\mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(\mathsf{w})$ as the set of identifiers of records that contain an entry matching w, along with the attribute-value pairs corresponding to the join attributes for the same record. In other words, $\mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(\mathsf{w})$ is a set of the form:

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

We refer to the collection of sets $\{\mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(\mathsf{w}_\ell)\}_{\mathsf{w}_\ell \in \mathsf{W}_i}$ as the "inverted join index" for the table Tab_i.

Join query. A *join query* over a pair of tables Tab_{i_1} and Tab_{i_2} with corresponding attribute-value pair sets W_1 and W_2, respectively, is specified by a tuple

$$q = (i_1, i_2, \mathsf{w}_1, \mathsf{w}_2, \mathsf{attr}^*),$$

where $\mathsf{w}_1 \in \mathsf{W}_1$, $\mathsf{w}_2 \in \mathsf{W}_2$, and attr^* is a special "join attribute" that defines the join relation between the tables Tab_{i_1} and Tab_{i_2} for the query q.

We write $\mathsf{DB}(q)$ to be the set of tuples of the form $(\mathsf{ind}_1, \mathsf{ind}_2)$ that "satisfy" the query q, where ind_1 and ind_2 are identifiers corresponding to records in the tables Tab_{i_1} and Tab_{i_2}, respectively. Formally, this means that for each $(\mathsf{ind}_1, \mathsf{ind}_2) \in \mathsf{DB}(q)$, the following conditions hold simultaneously:

$$((\mathsf{ind}_1, \mathsf{w}_1) \in \mathsf{Tab}_{i_1}) \wedge ((\mathsf{ind}_2, \mathsf{w}_2) \in \mathsf{Tab}_{i_2}),$$

$$\exists \gamma \text{ s.t. } ((\mathsf{ind}_1, \langle \mathsf{attr}^*, \gamma \rangle) \in \mathsf{Tab}_{i_1}) \wedge ((\mathsf{ind}_2, \langle \mathsf{attr}^*, \gamma \rangle) \in \mathsf{Tab}_{i_2})$$

2.2 SSE Syntax and Security Model

In this section, we formally define searchable symmetric encryption (SSE). Before presenting the formal definition, we present certain assumptions we make in the rest of the paper.

- In the rest of the paper, we assume that any plaintext record is identified by its index ind while, the corresponding encrypted version of the record is identified by a "randomized index" rind (computed as a pseudorandom mapping applied on the original index ind).

– We also assume that the output from the SSE protocol for a given search query are the indices (or identifiers) ind corresponding to the records that satisfy the query. A client program can then use these indices to retrieve the encrypted records and decrypt them. We adopt this formulation because it allows us to decouple the storage of payloads (which can be done in a variety of ways, with varying types of leakage) from the storage of metadata, which is the focus of our protocol (e.g., a client may retrieve the encrypted records from the same server running the query or from a different server, or may only retrieve records not previously cached, etc.)

We note here that a similar formulation is used by almost all existing works on SSE, and more generally structured encryption [CGKO06,CK10,CJJ$^+$13,CJJ$^+$14].

Formal Definition of SSE. A *searchable symmetric encryption (SSE) scheme* Π consists of an algorithm $\mathsf{EDBSetup}$ and a protocol Search between the client and server, all fitting the following syntax:

– $\mathsf{EDBSetup}$ takes as input a database DB, and outputs a secret key K along with an encrypted database EDB.
– The Search protocol is between a *client* and *server*, where the client takes as input the secret key K and a query q and the server takes as input EDB. At the end, the client outputs a set of identifiers, and the server has no output.

Correctness. We say that an SSE scheme is *correct* if for all inputs DB and queries q, if $(K, \mathsf{EDB}) \overset{\$}{\leftarrow} \mathsf{EDBSetup}(\mathsf{DB})$, after running Search with client input (K, q) and server input EDB, the client outputs the set of indices $\mathsf{DB}(q)$.

Adaptive Security of SSE. We recall the semantic security definitions of SSE from [CGKO06,CK10]. The definition is parameterized by a *leakage function* $\mathcal{L}$, which describes what an adversary (the server) is allowed to learn about the database and queries. Formally, security says that the server's view during an adaptive attack (where the server selects the database and queries) can be simulated given only the output of $\mathcal{L}$.

Definition 1. *Let* $\Pi = (\mathsf{EDBSetup}, \mathsf{Search})$ *be an SSE scheme and let* $\mathcal{L}$ *be a stateful algorithm. For algorithms* $\mathcal{A}$ *(denoting the adversary) and* S *(denoting a simulator), we define the experiments (algorithms)* $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ *and* $\mathbf{Ideal}^{\Pi}_{\mathcal{A},S}(\lambda)$ *as follows:*

$\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ *:* $\mathcal{A}(1^\lambda)$ *chooses* DB. *The experiment then runs*

$$(K, \mathsf{EDB}) \leftarrow \mathsf{EDBSetup}(\mathsf{DB}),$$

and gives EDB *to* $\mathcal{A}$. *Then* $\mathcal{A}$ *repeatedly chooses a query* q. *To respond, the game runs the* Search *protocol with client input* (K, q) *and server input* EDB *and gives the transcript and client output to* $\mathcal{A}$. *Eventually* $\mathcal{A}$ *returns a bit that the game uses as its own output.*

Ideal$^{\Pi}_{\mathcal{A},S}(\lambda)$ *: The game initializes a counter* $\mathsf{cnt} = 0$ *and an empty list* $\mathbf{q}$. $\mathcal{A}(1^\lambda)$ *chooses* DB*. The experiment runs* $\mathsf{EDB} \leftarrow S(\mathcal{L}(\mathsf{DB}))$ *and gives* EDB *to* $\mathcal{A}$*. Then* $\mathcal{A}$ *repeatedly chooses a query* q*. To respond, the game records this as* $\mathbf{q}[i]$*, increments* i*, and gives to* $\mathcal{A}$ *the output of* $S(\mathcal{L}(\mathsf{DB}, \mathbf{q}))$*. (Note that here,* $\mathbf{q}$ *consists of all previous queries in addition to the latest query issued by* $\mathcal{A}$*.) Eventually* $\mathcal{A}$ *returns a bit that the game uses as its own output.*

We say that Π *is* $\mathcal{L}$*-semantically-secure against adaptive attacks if for all adversaries* $\mathcal{A}$ *there exists an algorithm* S *such that*

$$| \Pr[\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\Pi}_{\mathcal{A},S}(\lambda) = 1] | \leq \mathrm{neg}(\lambda).$$

We note that in the security analysis of our SSE schemes we include the client's output, the set of indices $\mathsf{DB}(\psi(\bar{w}))$, in the adversary's view in the real game, to model the fact that these ind's will be used for retrieval of encrypted record payloads.

Selective Security of SSE. We also consider a weaker version of *selective* security for SSE that is identical to the adaptive security definition except that: (a) in the real world experiment, the adversary $\mathcal{A}$ does not get to choose its queries adaptively, but is required to specify all such queries non-adaptively at the beginning of the protocol along with the plaintext database DB, and receives EDB and the transcript and client output corresponding to each of its queries together at the end of the experiment. Also, (b) in the ideal world experiment, the adversary $\mathcal{A}$ directly receives as output the final response of a non-adaptive simulator S, computed as $S(\mathcal{L}(\mathsf{DB}, \{\mathbf{q}[N]\}_{i\in[Q]}))$, where Q is the total number of queries issued by the adversary $\mathcal{A}$ non-adaptively.

2.3 TSets

We recall the definition of syntax and security for a *tuple set*, or TSet. Intuitively, a TSet allows one to associate a list of fixed-sized data tuples with each attribute-value pair in the database, and later issue related tokens to retrieve these lists. We will use it in our SSE protocols for join queries as an "expanded inverted join index".

TSet **Syntax.** Formally, a TSet implementation $\Sigma = (\mathsf{TSetSetup}, \mathsf{TSetGetTag}, \mathsf{TSetRetrieve})$ will consist of three algorithms with the following syntax:

- $\mathsf{TSetSetup}$ takes as input $\mathbf{T} = (\mathbf{T}_1, \ldots, \mathbf{T}_N)$, where each $\mathbf{T}_i$ for $i \in [N]$ is an array of lists of equal-length bit strings indexed by the elements of W_i, and outputs (TSet, K_T).
- $\mathsf{TSetGetTag}$ takes as input the key K_T and a tuple (i, w) and outputs stag_i.
- $\mathsf{TSetRetrieve}$ takes as input TSet and stag_i, and returns a list of strings.

TSet **Correctness.** We say that Σ is *correct* if for all $\{\mathsf{W}_i\}_{i\in[N]}$, all $\mathbf{T} = (\mathbf{T}_1, \ldots, \mathbf{T}_N)$, and any $\mathsf{w} \in \mathsf{W}_i$, we have

$$\mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag}) = \mathbf{T}_i[\mathsf{w}],$$

when $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$ and $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, (i, \mathsf{w}))$.

Intuitively, $\mathbf{T}$ holds lists of tuples associated with attribute-value pairs and correctness guarantees that the TSetRetrieve algorithm returns the data associated with the given attribute-value pair.

TSet **Security.** The security goal of a TSet implementation is to hide as much as possible about the tuples in $\mathbf{T} = (\mathbf{T}_1, \ldots, \mathbf{T}_N)$ and the attribute-value pairs these tuples are associated to, except for the vectors $\mathbf{T}_i[\mathsf{w}_1], \mathbf{T}_i[\mathsf{w}_2], \ldots$ of tuples revealed by the client's queried attribute-value pairs $\mathsf{w}_1, \mathsf{w}_2, \ldots$. (For the purpose of TSet implementation we equate client's query with a single attribute-value pair.)

The formal definition of security is similar to that of keyword-search based SSE for single-keyword queries. Since the list of tuples associated to searched attribute-value pairs can be viewed as information provided to the server, this information is also provided to the simulator in the security definition below.

We parameterize the TSet security definition with a leakage function $\mathcal{L}_T$ that describes what else the adversary is allowed to learn by looking at the TSet and stag values. For most implementations this leakage will reveal something about the structure of $\mathbf{T}$, and consequently also the structure of DB.

Definition 2. *Let $\Sigma = (\mathsf{TSetSetup}, \mathsf{TSetGetTag}, \mathsf{TSetRetrieve})$ be a* TSet *implementation, and let $\mathcal{A}, S$ be an adversary and a simulator, and let $\mathcal{L}_T$ be a stateful algorithm. We define two games, $\mathbf{Real}^{\Sigma}_{\mathcal{A}}$ and $\mathbf{Ideal}^{\Sigma}_{\mathcal{A}}$ as follows.*

$\mathbf{Real}^{\Sigma}_{\mathcal{A}}(\lambda)$ *: $\mathcal{A}(1^\lambda)$ outputs $\{\mathsf{W}_i\}_{i\in[N]}, \mathbf{T} = (\mathbf{T}_1, \ldots, \mathbf{T}_N)$ with the above syntax. The game computes*

$$(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T}),$$

and gives TSet *to $\mathcal{A}$. Then $\mathcal{A}$ repeatedly issues queries $q \in \mathsf{W}$, and for each q the game gives* $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K, q)$ *to $\mathcal{A}$. Eventually, $\mathcal{A}$ outputs a bit, which the game also uses as its output.*

$\mathbf{Ideal}^{\Sigma}_{\mathcal{A},S}(\lambda)$ *: The game initializes a counter $i = 0$ and an empty list $\mathbf{q}$. $\mathcal{A}(1^\lambda)$ outputs $\{\mathsf{W}_i\}_{i\in[N]}, \mathbf{T} = (\mathbf{T}_1, \ldots, \mathbf{T}_N)$ as above. The game runs $\mathsf{TSet} \leftarrow S(\mathcal{L}_T(\mathbf{T}))$ and gives* TSet *to $\mathcal{A}$. Then $\mathcal{A}$ repeatedly issues queries $q \in \mathsf{W}$, and for each q the game stores q in $\mathbf{q}[i]$, increments i, and gives to $\mathcal{A}$ the output of $S(\mathcal{L}_T(\mathbf{T}, \mathbf{q}), \mathbf{T}[q])$. Eventually, $\mathcal{A}$ outputs a bit, which the game also uses as its output.*

*We say that Σ is a $\mathcal{L}_T$-*adaptively-secure TSet implementation *if for all adversaries $\mathcal{A}$ there exists an algorithm S such that*

$$|\Pr[\mathbf{Real}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\Sigma}_{\mathcal{A},S}(\lambda) = 1]| \leq \operatorname{neg}(\lambda).$$

3 Join Cross-Tags (JXT): SSE for Joins

In this section, we formally describe our new JXT protocol for searching over joins of tables in encrypted relational databases. The JXT protocol consists of two protocols:

– The $\mathsf{EDBSetup}$ protocol is a randomized algorithm executed locally at the client. This protocol takes as input the plaintext database and generates the encrypted database EDB, which is to be outsourced to the (untrusted) server. The encrypted database EDB consists of two data structures - the TSet and the XSet. The $\mathsf{EDBSetup}$ protocol also generates a secret key K, which is stored locally at the client and is used subsequently to generate query tokens.
– The Search protocol is used to execute queries over joins of encrypted tables in the encrypted database EDB. At a high level, it is a two-party protocol executed jointly by the client and the server, where the client's input is the query to be executed and the server's input is the encrypted database EDB. It consists of a single round of communication (i.e. a query message from the client to the server, followed by a response message from the server to the client). At the end of the protocol, the client is expected to learn the set of record indices (across the two tables) matching the join query.

We now expand in more details on how each of the aforementioned protocols function. In what follows, we assume that: (a) $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is a family of pseudorandom functions, (b) $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is an IND-CPA secure symmetric-key encryption algorithm with λ-bit keys, and (c) $\Sigma = (\mathsf{TSetSetup}, \mathsf{TSetGetTag}, \mathsf{TSetRetrieve})$ is a secure TSet as defined in Section 2.

3.1 The EDBSetup Algorithm of JXT

We now describe the $\mathsf{EDBSetup}$ algorithm of JXT. A summary of how the algorithm works appears in Figure 2.

We note that in JXT, each table Tab is processed independently; so we focus on the processing for a single table. Given a table Tab, let W denote the set of attribute-value pairs across this table Tab. Also, let $\{\mathsf{attr}_t^*\}_{t\in[T]}$ denote the set of T special attributes over which we allow the table Tab to be joined with other tables in the database. We begin by describing how the XSet component of the encrypted database is generated for a given table.

Generating the XSet Table-wise. For each record with identifier ind in the table Tab, let $\{\mathsf{w}_t^*\}_{t\in[T]}$ denote the set of attribute-value pairs for this record with identifier ind corresponding to the T special "join attributes". For each $t \in [T]$, the $\mathsf{EDBSetup}$ algorithm computes the values

$$\mathsf{xind}_t = F(K_I, t, \mathsf{ind}), \quad \mathsf{xw}_t = F(K_W, \mathsf{w}_t^*),$$

where $K_I, K_W \in \{0,1\}^\lambda$ are uniformly sampled keys for the PRF family F. Additionally, the $\mathsf{EDBSetup}$ algorithm computes the "cross-tag"

$$\mathsf{xtag}_t = \mathsf{xw}_t + \mathsf{xind}_t,$$

where xw_t and xind_t are as described above. The XSet corresponding to the table Tab is then populated with all such xtag values.

```
EDBSetup(DB)

1. Sample uniformly random keys $K_I, K_W, K_Z, K_{Z'}, K_{\mathsf{enc}}$ for the PRF $F$ and parse the database
   as $\mathsf{DB} = \{\mathsf{Tab}_i, \mathsf{W}_i\}_{i\in[N]}$.
2. For each table $\mathsf{Tab}_i$, proceed as follows:
   (a) For each pair of record index and join attribute-values of the form $(\mathsf{ind}, \{\mathsf{w}_t^*\}_{t\in[T]})$ in $\mathsf{Tab}_i$,
       build the set $\mathsf{XSet}[i]$ as follows:
       i. Set the following (for each $t \in [T]$):
          $\mathsf{xind}_t = F(K_I, t, \mathsf{ind}) \quad , \quad \mathsf{xw}_t = F(K_W, \mathsf{w}_t^*) \quad , \quad \mathsf{xtag}_t = \mathsf{xw}_t + \mathsf{xind}_t.$
       ii. Add the entries $\mathsf{xtag}_t$ (one for each $t \in [T]$) to $\mathsf{XSet}[i]$.
   (b) For each $\mathsf{w} \in \mathsf{W}_i$, build the tuple list $\mathbf{T}_i[\mathsf{w}]$ as follows:
       i. Set $z_0 = F(K_Z, \mathsf{w} \,\|\, 0) \quad , \quad z_0' = F(K_{Z'}, \mathsf{w} \,\|\, 0)$.
       ii. For all $(\mathsf{ind}, \{\mathsf{w}_t^*\}_{t\in[T]}) \in \mathsf{DB}^{\mathsf{Join}}_{\mathsf{Tab}_i}(\mathsf{w})$ in random order, initialize a counter $\mathsf{cnt} = 1$, and
           proceed as follows:
           A. Set $z_{\mathsf{cnt}} = F(K_Z, \mathsf{w} \,\|\, \mathsf{cnt}) \quad , \quad z_{\mathsf{cnt}}' = F(K_{Z'}, \mathsf{w} \,\|\, \mathsf{cnt})$.
           B. For each $t \in [T]$:
              – Set $\mathsf{xind}_t = F(K_I, t, \mathsf{ind})$, and $\mathsf{xw}_t = F(K_W, \mathsf{w}_t^*)$.
              – Set $y_t = \mathsf{xind}_t - (z_0 + z_{\mathsf{cnt}})$, and $y_t' = \mathsf{xw}_t - (z_0' + z_{\mathsf{cnt}}')$.
           C. Set $K_{\mathsf{enc},\mathsf{w}} = F(K_{\mathsf{enc}}, \mathsf{w})$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(K_{\mathsf{enc},\mathsf{w}}, \mathsf{ind})$.
           D. Append $(\mathsf{ct}, \{y_t, y_t'\}_{t\in[T]})$ to $\mathbf{t}$.
           E. Set $\mathsf{cnt} \leftarrow \mathsf{cnt} + 1$.
       iii. Set $\mathbf{T}_i[w] = \mathbf{t}$.
3. $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T}_1 \| \ldots \| \mathbf{T}_N)$.
4. Output the key $K = (K_I, K_W, K_Z, K_{Z'}, K_{\mathsf{enc}}, K_T)$ and $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$.
```

Fig. 2: The setup algorithm of Join Cross-Tags (JXT.EDBSetup). We assume that each record index ind in a table Tab_i is appended with the table number i.

Remark. Looking ahead, the XSet is used primarily for membership-testing, hence we can implement this using a Bloom filter to save storage (this is essentially similar to what is done for the XSet in the OXT protocol).

Generating the TSet Table-wise. We now describe how to generate the TSet component for the table Tab. For each attribute-value pair w in the set W for the table Tab, the EDBSetup algorithm does the following:

– It generates a pair of "padding elements" of the form

$$z_0 = F(K_Z, \mathsf{w} \,\|\, 0) \quad , \quad z_0' = F(K_{Z'}, \mathsf{w} \,\|\, 0),$$

where K_Z and $K_{Z'}$ are again uniformly sampled keys for the PRF family F.

– Suppose that the attribute-value pair w occurs in a record with identifier ind, and let $\{\mathsf{w}_t^*\}_{t\in[T]}$ denote the set of attribute-value pairs for this record with identifier ind corresponding to the T special "join attributes". To each such record, the EDBSetup algorithm assigns a *unique* counter value $\mathsf{cnt} \geq 1$

and computes the following additional "padding elements":

$$z_{\mathsf{cnt}} = F(K_Z, \mathsf{w} \,\|\, \mathsf{cnt}), \quad z'_{\mathsf{cnt}} = F(K_{Z'}, \mathsf{w} \,\|\, \mathsf{cnt}).$$

– In addition, for each $t \in [T]$, the $\mathsf{EDBSetup}$ algorithm computes

$$\mathsf{xind}_t = F(K_I, t, \mathsf{ind}), \quad \mathsf{xw}_t = F(K_W, \mathsf{w}^*_t),$$

$$y_t = \mathsf{xind}_t - (z_0 + z_{\mathsf{cnt}}), \quad y'_t = \mathsf{xw}_t - (z'_0 + z'_{\mathsf{cnt}}).$$

Remark. Note that xind_t and xw_t are generated identically as in the computation of the TSet. In fact, while we duplicate the generation of these elements for ease of understanding, in a real execution of the algorithm, these values can be generated exactly once and re-used for the generation of the XSet and the TSet.

– Finally, the $\mathsf{EDBSetup}$ algorithm computes the randomized index ct for the index ind as

$$K_{\mathsf{enc,w}} = F(K_{\mathsf{enc}}, \mathsf{w}), \quad \mathsf{ct} = \mathsf{Enc}(K_{\mathsf{enc,w}}, \mathsf{ind}),$$

where K_{enc} is again a uniformly sampled key for the PRF family F.

Overall, the TSet entry corresponding to the attribute-value pair w consists of an entry corresponding to each record ind containing w, where each such entry is a tuple of the form $(\mathsf{ct}, \{y_t, y'_t\}_{t\in[T]})$, generated as described above. The actual TSet is then generated using the secure T-Set implementation Σ.

3.2 The Search Protocol of JXT

We now describe how the Search protocol works on a join query of the form $q = \left(i, j, \mathsf{w}^{(1)}, \mathsf{w}^{(2)}, \mathsf{attr}^*_{i,j}\right)$, which essentially denotes a query over the join of the tables Tab_i and Tab_j, where the join is computed with respect to the special attribute $\mathsf{attr}^*_{i,j}$, which is a designated "join attribute" for both tables Tab_i and Tab_j. A concise summary of how the protocol works appears in Figure 3.

At a high level, the search protocol can be divided into three parts:

– **[Round-1 (Client → Server)]:** The client generates a "query message" and sends it across to the server.
– **[Round-2 (Server → Client)]:** The server generates a "response message" and sends it across to the client.
– **[Local Computation (Client)]:** The client performs some local computation to retrieve the final set of record identifiers matching the query.

We describe how each of these parts work.

[Round-1:] Query Message (Client → Server). The client sends to the server the table indices i and j, along with the join attribute $\mathsf{attr}^*_{i,j}$ over which the query is to be executed. The client also sends to the server the stag values $\mathsf{stag}^{(1)}$ and

Search protocol

1. The client has input the key K and a join query $q = (i, j, \mathsf{w}^{(1)}, \mathsf{w}^{(2)}, \mathsf{attr}^*_{i,j})$, and proceeds as follows:
   - Send to the server $(i, j, \mathsf{attr}^*_{i,j})$. Locally compute and store
   $$K_{\mathsf{enc},\mathsf{w}^{(1)}} = F(K_{\mathsf{enc}}, \mathsf{w}^{(1)}), \quad K_{\mathsf{enc},\mathsf{w}^{(2)}} = F(K_{\mathsf{enc}}, \mathsf{w}^{(2)}).$$
   - Send to the server $(\mathsf{stag}^{(1)}, \mathsf{stag}^{(2)})$, where
   $$\mathsf{stag}^{(1)} \leftarrow \mathsf{TSetGetTag}(K_T, (i, \mathsf{w}^{(1)})),\ \mathsf{stag}^{(2)} \leftarrow \mathsf{TSetGetTag}(K_T, (j, \mathsf{w}^{(2)})).$$
   - For $\mathsf{cnt}^{(1)} = 1, 2 \ldots$ and until server sends $\mathsf{stop}^{(1)}$, send to the server
   $$\mathsf{xjointoken}^{(1)}[\mathsf{cnt}^{(1)}] = F(K_{Z'}, \mathsf{w}^{(1)} \parallel \mathsf{cnt}^{(1)}) + F(K_Z, \mathsf{w}^{(2)} \parallel 0).$$
   - For $\mathsf{cnt}^{(2)} = 1, 2 \ldots$ and until server sends $\mathsf{stop}^{(2)}$, send to the server
   $$\mathsf{xjointoken}^{(2)}[\mathsf{cnt}^{(2)}] = F(K_{Z'}, \mathsf{w}^{(1)} \parallel 0) + F(K_Z, \mathsf{w}^{(2)} \parallel \mathsf{cnt}^{(2)}).$$

2. The server has input $(\mathsf{TSet}, \mathsf{XSet})$ and responds to the messages from the client as follows.
   (a) It sets:
   $$\mathbf{t}^{(1)} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, (i, \mathsf{stag}^{(1)})),\ \mathbf{t}^{(2)} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, (j, \mathsf{stag}^{(2)})).$$
   (b) For $\mathsf{cnt}^{(1)} = 1, \ldots, |\mathbf{t}^{(1)}|$, the server does the following:
      i. Retrieve $(\mathsf{ct}^{(1)}, \{y_t^{(1)}, {y'}_t^{(1)}\}_{t \in [T]})$ from the $\mathsf{cnt}^{(1)}$-th tuple in $\mathbf{t}^{(1)}$. Let ${y'}_{t^*}^{(1)}$ be the entry from among $\{{y'}_t^{(1)}\}_{t \in [T]}$ corresponding to the attribute $\mathsf{attr}^*_{i,j}$.
      ii. Set $\mathsf{xtoken}_{t^*}^{(1)} = \mathsf{xjointoken}^{(1)}[\mathsf{cnt}^{(1)}] + {y'}_{t^*}^{(1)}$.
   (c) When last tuple in $\mathbf{t}^{(1)}$ is reached, send $\mathsf{stop}^{(1)}$ to the client.
   (d) for $\mathsf{cnt}^{(2)} = 1, \ldots, |\mathbf{t}^{(2)}|$:
      i. Retrieve $(\mathsf{ct}^{(2)}, \{y_t^{(2)}, {y'}_t^{(2)}\}_{t \in [T]})$ from the $\mathsf{cnt}^{(2)}$-th tuple in $\mathbf{t}^{(2)}$. Let $y_{t^*}^{(2)}$ be the entry from among $\{y_t^{(2)}\}_{t \in [T]}$ corresponding to the attribute $\mathsf{attr}^*_{i,j}$.
      ii. Set $\mathsf{xtoken}_{t^*}^{(2)} = \mathsf{xjointoken}^{(2)}[\mathsf{cnt}^{(2)}] + y_{t^*}^{(2)}$.
   (e) Send $\mathsf{stop}^{(2)}$ to the client.
   (f) For $\mathsf{cnt}^{(1)} = 1, \ldots, |\mathbf{t}^{(1)}|$ and $\mathsf{cnt}^{(2)} = 1, \ldots, |\mathbf{t}^{(2)}|$:
   If $(\mathsf{xtoken}_{t^*}^{(1)} + \mathsf{xtoken}_{t^*}^{(2)}) \in \mathsf{XSet}[j]$, then send $(\mathsf{ct}^{(1)}, \mathsf{ct}^{(2)})$ to the client.

3. For each $(\mathsf{ct}^{(1)}, \mathsf{ct}^{(2)})$ received from the server, the client recovers and outputs:
$$\mathsf{ind}^{(1)} = \mathsf{Dec}(K_{\mathsf{enc},\mathsf{w}^{(1)}}, \mathsf{ct}^{(1)}), \quad \mathsf{ind}^{(2)} = \mathsf{Dec}(K_{\mathsf{enc},\mathsf{w}^{(2)}}, \mathsf{ct}^{(2)}),$$

Fig. 3: The search protocol of Join Cross-Tags (JXT.Search).

$\mathsf{stag}^{(2)}$, which allow the server to recover the TSet entries corresponding to the attribute value pairs $\mathsf{w}^{(1)}$ and $\mathsf{w}^{(2)}$, respectively. In addition, corresponding to each TSet entry for the attribute value pairs $\mathsf{w}^{(1)}$ and $\mathsf{w}^{(2)}$, the client sends across to the server a sequence of terms of the form

$$\mathsf{xjointoken}^{(1)}[1], \mathsf{xjointoken}^{(1)}[2], \ldots$$

$$\mathsf{xjointoken}^{(2)}[1], \mathsf{xjointoken}^{(1)}[2], \ldots$$

until the server sends the signals $\mathsf{stop}^{(1)}$ and $\mathsf{stop}^{(2)}$, respectively, indicating that there are no more TSet entries to process for either attribute-value pair. These terms are generated as follows: for a given counter value $\mathsf{cnt}^{(1)} \in \{1, 2, \ldots\}$, the term $\mathsf{xjointoken}^{(1)}[\mathsf{cnt}^{(1)}]$ is generated as:

$$\mathsf{xjointoken}^{(1)}[\mathsf{cnt}^{(1)}] = F(K_{Z'}, \mathsf{w}^{(1)} \,\|\, \mathsf{cnt}^{(1)}) + F(K_Z, \mathsf{w}^{(2)} \,\|\, 0).$$

Similarly, for a given counter value $\mathsf{cnt}^{(2)} \in \{1, 2, \ldots\}$, the term $\mathsf{xjointoken}^{(2)}[\mathsf{cnt}^{(2)}]$ is generated as:

$$\mathsf{xjointoken}^{(2)}[\mathsf{cnt}^{(2)}] = F(K_{Z'}, \mathsf{w}^{(1)} \,\|\, 0) + F(K_Z, \mathsf{w}^{(2)} \,\|\, \mathsf{cnt}^{(2)}).$$

[Round-2:] Response Message (Server $\rightarrow$ Client). The server uses the stag values sent across by the client to recover the TSet entries corresponding to the attribute-value pairs $\mathsf{w}^{(1)}$ and $\mathsf{w}^{(2)}$. More specifically:

– The server uses $\mathsf{stag}^{(1)}$ (received from the client as part of the first round message) to recover the TSet entries corresponding to the attribute-value pair $\mathsf{w}^{(1)}$ from the T-Set corresponding to table Tab_i. Suppose that each such entry is a tuple of the form

$$(\mathsf{ct}^{(1)}, \{y_t^{(1)}, {y'}_t^{(1)}\}_{t \in [T]}).$$

Also, let ${y'}_{t^*}^{(1)}$ be the entry from among $\{{y'}_t^{(1)}\}_{t \in [T]}$ corresponding to the attribute $\mathsf{attr}_{i,j}^*$ over which the query is being executed. The server computes

$$\mathsf{xtoken}_{t^*}^{(1)} = \mathsf{xjointoken}^{(1)}[\mathsf{cnt}^{(1)}] + {y'}_{t^*}^{(1)}.$$

– Similarly, the server uses $\mathsf{stag}^{(2)}$ (also received from the client as part of the first round message) to recover the TSet entries corresponding to the attribute-value pair $\mathsf{w}^{(2)}$ from the T-Set corresponding to table Tab_j. Suppose that each such entry is a tuple of the form

$$(\mathsf{ct}^{(2)}, \{y_t^{(2)}, {y'}_t^{(2)}\}_{t \in [T]}).$$

Again, let $y_{t^*}^{(2)}$ be the entry from among $\{y_t^{(2)}\}_{t \in [T]}$ corresponding to the attribute $\mathsf{attr}_{i,j}^*$ over which the query is being executed. The server computes

$$\mathsf{xtoken}_{t^*}^{(2)} = \mathsf{xjointoken}^{(2)}[\mathsf{cnt}^{(2)}] + y_{t^*}^{(2)}.$$

Now, for each such pair of TSet entries (where the first entry corresponds to the attribute-value pair $\mathsf{w}^{(1)}$ from the T-Set for table Tab_i, and the second corresponds to the attribute-value pair $\mathsf{w}^{(2)}$ from the T-Set for table Tab_j), the server computes a candidate xtag value of the form

$$\mathsf{xtag}^{(1,2)} = \mathsf{xtoken}_{t^*}^{(1)} + \mathsf{xtoken}_{t^*}^{(2)},$$

and checks the membership of $\mathsf{xtag}^{(1,2)}$ in the XSet corresponding to Tab_j.

– If the membership-test returns **true**, then the server infers that the pair of records match constitute a matching record in the join of the two tables; hence it sends back the corresponding randomized identifiers $(\mathsf{ct}^{(1)}, \mathsf{ct}^{(2)})$ to the client.
– If the membership test returns **false**, then the server discards the corresponding randomized identifiers.

Local Computation (Client): Finally, the client decrypts the set of randomized record identifiers (i.e., the tuples of the form $(\mathsf{ct}^{(1)}, \mathsf{ct}^{(2)})$) sent across by the client, and decrypts them to retrieve the set of plaintext record identifiers corresponding to the records matching the query q.

Realizing TSet and XSet. We note here that an implementation of JXT can use the same selectively/adaptively secure implementations of TSet (built from purely symmetric-key cryptographic primitives) as used by OXT [CJJ+13]. We also note here that one could equivalently use an encrypted multi-map (EMM) [CK10,KM17,KM19] instead of a TSet in JXT. Also note that during the search protocol, the server uses the XSet purely for membership-testing. This allows implementing the XSet using a Bloom filter, as in OXT. These observations provide evidence for the overall compatibility of JXT with OXT.

Correctness. We now formally argue that the JXT protocol is correct. More concretely, we state and prove the following theorem:

Theorem 1. *Assuming that* SKE *satisfies decryption correctness and* Σ *is a correct* TSet *implementation, the* JXT *protocol satisfies correctness.*

Proof. Consider a query of the form $q = (i, j, \mathsf{w}^{(1)}, \mathsf{w}^{(2)}, \mathsf{attr}^*_{i,j})$, and suppose that there exists an index-pair $(\mathsf{ind}_1, \mathsf{ind}_2) \in \mathsf{DB}(q)$. We now argue that the client recovers $(\mathsf{ind}_1, \mathsf{ind}_2)$ as an outcome of the Search protocol. To see this, observe the following. Since the aforementioned conditions hold true, the server must retrieve the following TSet entries corresponding to $\mathsf{w}^{(1)}$ and $\mathsf{w}^{(2)}$ (this follows from the correctness of the TSet implementation σ):

$$(\mathsf{ct}^{(1)}, \{y_t^{(1)}, {y'}_t^{(1)}\}_{t\in[T]}), \quad (\mathsf{ct}^{(2)}, \{y_t^{(2)}, {y'}_t^{(2)}\}_{t\in[T]}),$$

where $\mathsf{ct}^{(1)} = \mathsf{Enc}(K_{\mathsf{enc,w}}, \mathsf{ind}_1)$ and $\mathsf{ct}^{(2)} = \mathsf{Enc}(K_{\mathsf{enc,w}}, \mathsf{ind}_2)$, and letting ${y'}_{t^*}^{(1)}$ and $y_{t^*}^{(2)}$ be the respective entries corresponding to the attribute $\mathsf{attr}^*_{i,j}$,

$${y'}_{t^*}^{(1)} = F(K^{(2)}, \langle \mathsf{attr}^*_{i,j}, \gamma\rangle) - (F(K_{Z'}, \mathsf{w}^{(1)} \,\|\, 0) + F(K_{Z'}, \mathsf{w}^{(1)} \,\|\, \mathsf{cnt}^{(1)})),$$

$$y_{t^*}^{(2)} = F(K_I, t^*\mathsf{ind}_2) - (F(K_Z, \mathsf{w}^{(2)} \,\|\, 0) + F(K_Z, \mathsf{w}^{(2)} \,\|\, \mathsf{cnt}^{(2)})),$$

for some appropriate counter values $\mathsf{cnt}^{(1)}$ and $\mathsf{cnt}^{(2)}$. In addition, the client sends across to the server the values $\mathsf{xjointoken}^{(1)}[\mathsf{cnt}^{(1)}]$ and $\mathsf{xjointoken}^{(2)}[\mathsf{cnt}^{(2)}]$ where

$$\mathsf{xjointoken}^{(1)}[\mathsf{cnt}^{(1)}] = F(K_{Z'}, \mathsf{w}^{(1)} \,\|\, \mathsf{cnt}^{(1)}) + F(K_Z, \mathsf{w}^{(2)} \,\|\, 0),$$

$$\mathsf{xjointoken}^{(2)}[\mathsf{cnt}^{(2)}] = F(K_{Z'}, \mathsf{w}^{(1)} \,\|\, 0) + F(K_Z, \mathsf{w}^{(2)} \,\|\, \mathsf{cnt}^{(2)}).$$

Consequently, as per the Search protocol, the server computes

$$\mathsf{xtoken}_{t^*}^{(1)} = F(K^{(2)}, \langle \mathsf{attr}_{i,j}^*, \gamma\rangle) - F(K_{Z'}, \mathsf{w}^{(1)} \,\|\, 0) + F(K_Z, \mathsf{w}^{(2)} \,\|\, 0),$$

and

$$\mathsf{xtoken}_{t^*}^{(2)} = F(K_I, t^*\mathsf{ind}_2) - F(K_Z, \mathsf{w}^{(2)} \,\|\, 0) + F(K_{Z'}, \mathsf{w}^{(1)} \,\|\, 0).$$

Next, the server computes the candidate xtag as

$$\mathsf{xtag}^{(1,2)} = \mathsf{xtoken}_{t^*}^{(1)} + \mathsf{xtoken}_{t^*}^{(2)} = F(K_I, t^*\mathsf{ind}_2) + F(K^{(2)}, \langle \mathsf{attr}_{i,j}^*, \gamma\rangle).$$

Note that this is nothing but the xtag corresponding to the index-attribute value pair $(\mathsf{ind}_2, \mathsf{w}^* = \langle \mathsf{attr}_{i,j}^*, \gamma\rangle)$. Finally, assuming that the symmetric-key encryption scheme SKE satisfies correctness of decryption, the client recovers the index-pair $(\mathsf{ind}_1, \mathsf{ind}_2)$. A similar argument can be used to show that the client does not recover any index-pair $(\mathsf{ind}_1', \mathsf{ind}_2') \notin \mathsf{DB}(q)$. This completes the proof of correctness for the JXT protocol.

On Bloom Filter and False Positives. We point out that using a Bloom filter to realize the XSet data structure potentially introduces false positives. The rate of such false positives can be programmed by setting the parameters of the Bloom filter, thus yielding a tradeoff between storage and false positive rate. As an alternative to Bloom filter, one could use any non-lossy data structure that allows checking for set-membership. This would prevent false positives, albeit at the cost of some extra storage at the server end.

4 Complexity Analysis of JXT

In this section, we analyze the asymptotic complexity of JXT.

Storage Overhead. We first discuss the storage overhead for each table. Recall that in JXT, the TSet and XSet for the encrypted database are built table-wise. Hence, the total storage overhead for JXT is essentially the sum of the overheads for each individual table. Suppose that a table Tab has a total of n attributes, with $T \leq n$ amongst these being "join attributes"; i.e. attributes over which the table can be joined with other tables in the database. Also, assume that Tab has a total of m records (equivalently, rows). We enumerate below the number of entries in the TSet and XSet corresponding to Tab.

Recall that for each attribute-value pair w in the set W for the table Tab, the TSet stores as many entries as the number of records containing the attribute-value pair w, where each such entry is a tuple of the form: $(\mathsf{ct}, \{y_t, y'_t\}_{t\in[T]})$. In other words, each entry is a collection of $(2T+1)$ objects. Hence, the total number of TSet entries for Tab is $\sum_{\mathsf{w}\in\mathsf{W}}(2T+1)|\mathsf{DB}_{\mathsf{Tab}}(\mathsf{w})|$. But note that $\sum_{\mathsf{w}\in\mathsf{W}} |\mathsf{DB}_{\mathsf{Tab}}(\mathsf{w})| = m\cdot n$, where m and n are the total number of records and attributes in the table Tab, respectively. Hence, $|\mathsf{TSet}(\mathsf{Tab})| = m\cdot n\cdot(2T+1)$. In other words, the TSet incurs an $O(T)$-fold overhead over the storage required for the plaintext table Tab.

Next, recall that the XSet for the table Tab has T entries corresponding to each record index ind. More concretely, for each record with identifier ind in the table Tab, let $\{\mathsf{w}^*_t\}_{t\in[T]}$ denote the set of attribute-value pairs for this record with identifier ind corresponding to the T special "join attributes". Then, for each $t\in[T]$, the $\mathsf{EDBSetup}$ algorithm stores a unique xtag_t entry corresponding to the pair $(\mathsf{ind}, \mathsf{w}^*_t)$. Thus, we have $|\mathsf{XSet}(\mathsf{Tab})| = m\cdot T$.

We note, however, that the XSet is implemented using a Bloom filter; consequently, the storage overhead for the XSet is significantly lower. As in OXT, we expect the overhead for the XSet in JXT to be low enough for the server to be able to store it in the RAM. The TSet will typically be stored on the disk.

Computational and Communication Overheads. We now present an asymptotic analysis for the computational and communication overheads when executing a search query over the joins of two tables Tab_1 and Tab_2. Suppose that the query involves two attribute-value pairs w_1 and w_2, and is to be executed over the join of Tab_1 and Tab_2 w.r.t. the attribute attr^*.

Computational Overhead (Client). The client computes the stag values corresponding to w_1 and w_2 using $O(1)$ invocations of the stag-generation algorithm for the TSet (the exact overhead depends on the implementation of the TSet; however, for efficient implementations such as the one for OXT [CJJ+13], this is a constant overhead). The client also computes $\mathsf{xjointoken}^{(1)}$ and $\mathsf{xjointoken}^{(2)}$ values; the number of such computations is $|\mathsf{DB}_{\mathsf{Tab}_1}(\mathsf{w}_1)| + |\mathsf{DB}_{\mathsf{Tab}_2}(\mathsf{w}_2)|$. Hence, the comp. overhead is $O(|\mathsf{DB}_{\mathsf{Tab}_1}(\mathsf{w}_1)| + |\mathsf{DB}_{\mathsf{Tab}_2}(\mathsf{w}_2)|)$.

Computational Overhead (Server). The server's computation can be broadly divided into two categories: (a) TSet lookups (using the stag values sent across by the client), and (b) xtag computations and membership-checks. The total number of TSet lookups performed by the server corresponding to w_1 and w_2 is again $\mathsf{DB}_{\mathsf{Tab}_1}(\mathsf{w}_1)| + |\mathsf{DB}_{\mathsf{Tab}_2}(\mathsf{w}_2)$. The number of xtag computations is larger; in particular, the server computes (and checks membership of) a candidate xtag entry corresponding to each pair $(\mathsf{xjointoken}^{(1)}[\mathsf{cnt}^{(1)}], \mathsf{xjointoken}^{(2)}[\mathsf{cnt}^{(2)}])$. Hence, the computational overhead at the server is $O(|\mathsf{DB}_{\mathsf{Tab}_1}(\mathsf{w}_1)|\cdot|\mathsf{DB}_{\mathsf{Tab}_2}(\mathsf{w}_2)|)$. Note that this computational overhead is unavoidable since in the worst case, we have $|\mathsf{DB}(q)| = |\mathsf{DB}_{\mathsf{Tab}_1}(\mathsf{w}_1)|\cdot|\mathsf{DB}_{\mathsf{Tab}_2}(\mathsf{w}_2)|$, and the server must perform at least as much computation as is required to compute and send across to the client the final result set pertaining to the join query.

Communication Overhead. The message from the client to the server consists of $O(\mathsf{DB}_{\mathsf{Tab}_1}(\mathsf{w}_1)| + \mathsf{DB}_{\mathsf{Tab}_2}(\mathsf{w}_2)|)$ terms, while the message from the server to the client consists of $|\mathsf{DB}(q)|$ terms. Hence, the overall communication complexity is $O(|\mathsf{DB}_{\mathsf{Tab}_1}(\mathsf{w}_1)| + |\mathsf{DB}_{\mathsf{Tab}_2}(\mathsf{w}_2)| + |\mathsf{DB}(q)|)$.

Bloom Filter Configuration. We note here that the configuration of the Bloom filter used is expected to influence the performance of an actual implementation of JXT. We propose using Bloom filter configurations similar to those used in implementations of OXT reported in prior work [CJJ+13,CJJ+14].

5 Leakage Profile and Security of JXT

In this section, we formally describe the leakage profile of our JXT protocol(i.e. leakage to the server) for join queries, and prove its security with respect to this leakage profile.

5.1 The Leakage Profile of JXT

We represent a sequence of Q join queries by $\mathbf{q} = (\mathbf{i}_1, \mathbf{i}_2, \mathbf{s}_1, \mathbf{s}_2, \textbf{attr}^*)$, where an individual join query is represented (as per the definition of join queries introduced in Section 2) as a five-tuple $\mathbf{q}[\ell] = (\mathbf{i}_1[\ell], \mathbf{i}_2[\ell], \mathbf{s}_1[\ell], \mathbf{s}_2[\ell], \textbf{attr}^*[\ell])$. The leakage profile of JXT for such a sequence of join queries is a tuple of the form $\mathcal{L} = (\mathbf{n}, \mathsf{RP}, \mathsf{EP}_1, \mathsf{EP}_2, \mathsf{SP}_1, \mathsf{SP}_2, \mathsf{JD}, \mathsf{IP})$ where:

- $\mathbf{n}$ is an N-sized list, where for each $i \in [N]$, $\mathbf{n}[i]$ represents the total number of occurrences of all attribute-value pairs in W_i across records in table Tab_i.
- RP is the result pattern leakage, i.e. the set of records matching each query. Formally, we represent RP as a Q-sized list, where for each $\ell \in [Q]$, we have $\mathsf{RP}[\ell] = \mathsf{DB}(\mathbf{q}[\ell])$. Here, $\mathsf{DB}(q)$ for $q = \mathbf{q}[\ell]$ is as defined in Section 2.
- EP_1 is the equality pattern over $\mathbf{s}_1$ indicating which queries have equal attribute-value pairs in the *first* coordinate. Formally, we represent EP_1 as a $Q \times Q$ table with entries in $\{0, 1\}$, where $\mathsf{EP}_1[\ell, \ell'] = 1$ if $\mathbf{s}_1[\ell] = \mathbf{s}_1[\ell']$, and 0 otherwise.
- Similarly, EP_2 is the equality pattern over $\mathbf{s}_2$ indicating which queries have equal attribute-value pairs in the *second* coordinate. Formally, we represent EP_2 as a $Q \times Q$ table with entries in $\{0, 1\}$, where $\mathsf{EP}_2[\ell, \ell'] = 1$ if $\mathbf{s}_2[\ell] = \mathbf{s}_2[\ell']$, and 0 otherwise.
- SP_1 is the size pattern over $\mathbf{s}_1$, i.e. the number of records matching the *first* attribute-value pair in each join query. Formally, we represent SP_1 as a Q-sized list, where for each $\ell \in [Q]$, we have $\mathsf{SP}_1[\ell] = |\mathsf{DB}_{\mathsf{Tab}_{\mathbf{i}_1[\ell]}}(\mathbf{s}_1[\ell])|$.
- Similarly, SP_2 is the size pattern over $\mathbf{s}_2$, i.e. the number of records matching the *second* attribute-value pair in each join query. Formally, we represent SP_2 as a Q-sized list, where for each $\ell \in [Q]$, we have $\mathsf{SP}_2[\ell] = |\mathsf{DB}_{\mathsf{Tab}_{\mathbf{i}_2[\ell]}}(\mathbf{s}_2[\ell])|$.

– JD is the join attribute distribution pattern over $\mathbf{s}_1$, which is represented as a collection of Q multi-sets. The ℓ^{th} entry in this collection, i.e., $\mathsf{JD}[\ell]$ is a multi-set of (global) randomized encodings of the join attribute values corresponding to the join attribute $\mathbf{attr}^*[\ell]$ in the records matching the attribute-value pair $\mathbf{s}_1[\ell]$ in the table $\mathsf{Tab}_{\mathbf{i}_1[\ell]}$. More formally, for each $\ell \in [Q]$, we have the multi-set [10]

$$\mathsf{JD}[\ell] = \left\{\mathsf{encode}(\mathsf{val}^*) : (\mathsf{ind}, \mathbf{s}_1[\ell]) \in \mathsf{Tab}_{\mathbf{i}_1[\ell]} \textbf{ and } (\mathsf{ind}, \langle \mathbf{attr}^*[\ell], \mathsf{val}^* \rangle) \in \mathsf{Tab}_{\mathbf{i}_1[\ell]}\right\}.$$

– IP is the conditional intersection pattern, which is a $Q \times Q$ table with entries defined as explained next. For each $\ell, \ell' \in [Q]$, $\mathsf{IP}[\ell, \ell']$ is an empty set if one of the following conditions holds:

- $(\mathbf{i}_1[\ell], \mathbf{i}_2[\ell], \mathbf{attr}^*[\ell]) \neq (\mathbf{i}_1[\ell'], \mathbf{i}_2[\ell'], \mathbf{attr}^*[\ell'])$.
- $\mathsf{JD}[\ell] \cap \mathsf{JD}[\ell']$ is empty.

Otherwise, $\mathsf{IP}[\ell, \ell']$ is non-empty, and is defined as the intersection of all record identifiers matching the keywords $\mathbf{s}_2[\ell]$ and $\mathbf{s}_2[\ell']$ in the table $\mathsf{Tab}_{\mathbf{i}_2[\ell]}$. More formally, we have $\mathsf{IP}[\ell, \ell'] = \mathsf{DB}_{\mathsf{Tab}_{\mathbf{i}_2[\ell]}}(\mathbf{s}_2[\ell]) \cap \mathsf{DB}_{\mathsf{Tab}_{\mathbf{i}_2[\ell]}}(\mathbf{s}_2[\ell'])$.

5.2 Discussion on Leakage Components and Comparison with OXT

In this section, we present a discussion on the various components in the leakage profile of JXT, and also compare the same with OXT. Note that one fundamental difference between JXT and OXT is that while JXT supports queries over joins of multiple tables, OXT only supports "unilateral queries", where each such query is defined over a single table. This difference manifests in subtle distinctions between the leakage profiles for JXT and OXT, as described below.

The n-Leakage. Suppose that a table Tab_i has a total of n_i attributes (equivalently, columns) and a total of m_i records (equivalently, rows). We note here that $\mathbf{n}[i]$ is nothing but $n_i \cdot m_i$, i.e., the total number of entries in the table. We note that this information (or an upper bound thereof) is leaked by almost all existing SSE schemes in the literature with efficient search capabilities [CGKO06,CJJ$^+$13,CJJ$^+$14,LPS$^+$18], including OXT.

Result Pattern. The RP leakage allows the server to learn the set of identifiers corresponding to records in the result set for the query. Such a leakage is considered benign and is incurred by nearly all existing SSE schemes (notably [CGKO06,CK10,CJJ$^+$13,CJJ$^+$14]), including OXT. However, one subtle difference with OXT is that in JXT, the RP leakage spans across multiple tables, while in OXT, the RP leakage is confined to a single table. This, of course, is a direct consequence of the fact that JXT handles queries over the join of multiple tables, which OXT does not support.

Remark. We note here that our analysis of the result pattern leakage of JXT is rather conservative; an astute reader may observe that during the Search protocol in JXT, the server does not learn the actual plaintext identifiers for

[10] Note that a multi-set additionally reveals the frequency of each entry.

the records in the result set; it only learns the randomized/encrypted versions of these identifiers, which are then locally decrypted at the client.

Equality and Size Patterns. The EP leakage reveals repetitions of attribute-value pairs across join queries (including the "coordinate" of the join query where the repetition occurs), while the SP leakage reveals the individual frequency of each attribute-value pair in a given join query. The EP leakage can be mitigated by having more than one TSet entry per attribute-value pair, and the client using stag values that point to different entries for the same attribute-value pair across multiple queries, while the SP leakage can be potentially mitigated by artificially "padding" the number of TSet entries corresponding to each attribute-value pair; this would leak an upper bound rather than the exact frequency.

The EP and SP leakage can be viewed as consequences of our strategy of avoiding join pre-computations in the setup phase (and the corresponding blowup in storage overheads); since JXT processes each table individually rather than pre-computing their joins, processing a query over the join of two tables inevitably requires some independent searches over the individual TSet entries for each table. In particular, the EP and SP leakage of JXT are conceptually similar to the EP and SP leakage in OXT, albeit with the difference that in OXT, a unilateral search query over two conjuncts (referred to in OXT as the s-term and the x-term) incurs these leakage for only one of the terms (the s-term). We view the additional leakage in JXT as a necessary trade-off for the additional capability of handling join queries (or more concretely, existential quantifiers over the join attributes) with comparable efficiency.

Join Attribute Distribution Pattern. The JD leakage is new to JXT and is a direct consequence of the fact that it handles queries over joins of tables. For a given query over the join of two tables with respect to a common attribute (say attr^*), it reveals the frequency distribution of values taken by attr^* across records matching the attribute-value pair in the "first slot". The extent of this leakage depends on the "entropy" of the join attribute in the first table. For example, consider the case where the join-attribute is a primary key or a "high-entropy" key in the first table. In this case, it is likely to take a unique value for each record, and hence the JD leakage is essentially query-invariant (as each possible value occurs with frequency close to 1), and hence, benign. Thus the JD leakage can be mitigated by planning join queries (i.e., by ordering the attribute-value pairs in the "first" and "second" slots) such that join attribute is a primary/"high-entropy" key in the first table.

Conditional Intersection Pattern. The IP leakage of JXT is quite subtle; for a pair of queries over the join of the *same* tables over the *same* common attribute attr^*, it reveals the intersection of records matching the attribute-value pairs in the "second slot" provided that the attribute-value pairs in the "first slot" have matching records with identical $\langle \mathsf{attr}^*, \mathsf{val}^* \rangle$ entries. This leakage is conditioned on the fact that the attribute-value pairs in the "first slot" have such matching records; if such matching records do not exist, then this leakage is empty.

We note that the IP leakage is essentially guaranteed to be empty in either of the following scenarios: (a) the join-attribute is a primary key or a "high-entropy" key in the first table, in which case, it is likely to take a unique value for each record, or (b) the attribute-value pairs in the "second slot" share the same attribute but different values, in which case, they cannot match with the same record. In particular, similar to the JD leakage, the IP leakage can also be mitigated by planning join queries such that join attribute is a primary/"high-entropy" key in the first table. This bears similarities with the IP leakage of OXT, where the leakage can be minimized by re-arranging the conjuncts in each unilateral query such that the s-term has low frequency.

To summarize, the overall leakage profile of JXT bears many similarities with the leakage profile of OXT, and can be made benign in practice by simple query-planning strategies that do not compromise on practical search performance.

5.3 The Security Theorems for JXT

In this section, we state the theorems for the selective and adaptive security of JXT.

Selective Security of JXT. Let $\mathcal{L}$ be the leakage profile of JXT as described in Section 5.1. We state the following theorem.

Theorem 2. *Assuming that F is a secure PRF family,* SKE *is an IND-CPA secure symmetric-key encryption scheme, and Σ is a* **n***-selectively secure* TSet *implementation, the* JXT *protocol is $\mathcal{L}$-semantically simulation-secure against selective attacks.*

Proof Overview. We defer a detailed proof of this theorem to the full version of our paper [JP21]. The proof of selective security proceeds via a sequence of games between a simulator S and an adversary $\mathcal{A}$, where the first game is identical to the "ideal-world" game played between the simulator and the adversary $\mathcal{A}$ as described in Definition 1, while the final game is identical to the "real-world" game played between the simulator S and the adversary $\mathcal{A}$. We establish formally that the view of the adversary $\mathcal{A}$ in each pair of consecutive games is computationally indistinguishable.

The crux of our selective security proof is that the simulator for JXT can initialize the XSet to uniformly random elements, and program the outputs of the PRFs accordingly while making sure that the search tokens corresponding to a given join query are generated in a consistent manner. The programming is done given the leakage of JXT corresponding to the various search queries. Additionally, the simulator for JXT can directly invoke the simulator for the selectively secure TSet to simulate the TSet entries at setup and the corresponding stag values during searches. We refer to the full version of our paper [JP21] for the detailed description of the simulator, as well as for descriptions of the hybrids that allows us to prove the indistinguishability of this simulation from a real execution of JXT.

Adaptive Security of JXT. Again, let $\mathcal{L}$ be the leakage profile of JXT as described in Section 5.1.

Theorem 3. *Assuming that F is a secure PRF family,* SKE *is an IND-CPA secure symmetric-key encryption scheme, and Σ is a* $\mathbf{n}$*-adaptively secure* TSet *implementation, the* JXT *protocol is $\mathcal{L}$-semantically simulation-secure against adaptive attacks.*

Proof Overview. We again defer a detailed proof of this theorem to the full version of our paper [JP21]. In our adaptive proof of security, we assume an instantiation of adaptively secure TSet in the standard model. While the original construction of TSet [CJJ+13] requires random oracles for adaptive security, the authors of [CJJ+13] also discuss an alternative instantiation of adaptively secure TSets in the standard model without incurring additional rounds of communication. The idea is to send the actual addresses in the TSet (i.e., the outputs of PRF evaluation) directly to the server instead of sending the PRF key (or the stag), and allowing the server to compute PRF outputs on its own. In the context of our JXT protocol, using the standard model instantiation of TSet increases the communication overhead for the TSet component, but not the (asymptotic) communication overhead for the overall search protocol.

Acknowledgments

We thank the anonymous reviewers of IACR ASIACRYPT 2022 for their helpful comments and suggestions.

References

BKM20. Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *NDSS 2020*, 2020.

CGKO06. Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 2006*, pages 79–88, 2006.

CGPR15. David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM CCS 2015*, pages 668–679, 2015.

CJJ$^+$13. David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO 2013*, pages 353–373, 2013.

CJJ$^+$14. David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*, 2014.

CK10. Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT 2010*, pages 577–594, 2010.

CNR21. David Cash, Ruth Ng, and Adam Rivkin. Improved structured encryption for SQL databases via hybrid indexing. In *ACNS 2021*, pages 480–510, 2021.

CZH$^+$13. Cheng-Kang Chu, Wen Tao Zhu, Jin Han, Joseph K. Liu, Jia Xu, and Jianying Zhou. Security concerns in popular cloud storage services. *IEEE Pervasive Computing*, 12(4):50–57, 2013.

Gen09. C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM STOC'09*, pages 169–178, 2009.

GO96. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

Goh03. Eu-Jin Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

IKK12. Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*, 2012.

JJK$^+$13. Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In *ACM CCS 2013*, pages 875–888, 2013.

JP21. Charanjit S. Jutla and Sikhar Patranabis. Efficient searchable symmetric encryption for join queries (full version). *IACR Cryptol. ePrint Arch.*, page 1471, 2021.

KM17. Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT 2017*, pages 94–124, 2017.

KM18. Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. In *ASIACRYPT 2018*, pages 149–180, 2018.

KM19. Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In *EUROCRYPT 2019*, pages 183–213, 2019.

LPS$^+$18. Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shifeng Sun, Dongxi Liu, and Cong Zuo. Result pattern hiding searchable encryption for conjunctive queries. In *ACM CCS 2018*, pages 745–762, 2018.

NKW15. Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM CCS 2015*, pages 644–655, 2015.

PM21. Sikhar Patranabis and Debdeep Mukhopadhyay. Forward and backward private conjunctive searchable symmetric encryption. In *NDSS 2021*, 2021.

PRZB11a. Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *ACM SOSP 2011*, pages 85–100, 2011.

PRZB11b. Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP 2011*, pages 85–100, 2011.

SWP00. Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE S&P 2000*, pages 44–55, 2000.

ZKP16. Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium 2016*, pages 707–720, 2016.