Forward and Backward Private Conjunctive Searchable Symmetric Encryption

Sikhar Patranabis
ETH Zürich
sikhar.patranabis@inf.ethz.ch

Debdeep Mukhopadhyay
IIT Kharagpur
debdeep@cse.iitkgp.ac.in

***Abstract*—Dynamic searchable symmetric encryption (SSE) supports updates and keyword searches in tandem on outsourced symmetrically encrypted data, while aiming to minimize the information revealed to the (untrusted) host server. The literature on dynamic SSE has identified two crucial security properties in this regard - *forward* and *backward* privacy. Forward privacy makes it hard for the server to correlate an update operation with previously executed search operations. Backward privacy limits the amount of information learnt by the server about documents that have already been deleted from the database.**

To date, work on forward and backward private SSE has focused mainly on single keyword search. However, for any SSE scheme to be truly practical, it should at least support conjunctive keyword search. In this setting, most prior SSE constructions with sub-linear search complexity do not support dynamic databases. The only exception is the scheme of Kamara and Moataz (EUROCRYPT'17); however it only achieves forward privacy. Achieving *both* forward and backward privacy, which is the most desirable security notion for any dynamic SSE scheme, has remained open in the setting of conjunctive keyword search.

In this work, we develop the first forward and backward private SSE scheme for conjunctive keyword searches. Our proposed scheme, called Oblivious Dynamic Cross Tags (or ODXT in short) scales to very large arbitrarily-structured databases (including both attribute-value and free-text databases). ODXT provides a realistic trade-off between performance and security by efficiently supporting fast updates and conjunctive keyword searches over very large databases, while incurring only moderate access pattern leakages to the server that conform to existing notions of forward and backward privacy. We precisely define the leakage profile of ODXT, and present a detailed formal analysis of its security. We then demonstrate the practicality of ODXT by developing a prototype implementation and evaluating its performance on real world databases containing millions of documents.

I. Introduction

The advent of cloud computing potentially allows individuals and organizations to outsource storage and processing of large volumes of data to third party servers. However, this leads to privacy concerns - clients typically do not trust service providers to respect the confidentiality of their data [13]. This lack of trust is often fortified by threats from malicious insiders and external attackers.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23116
www.ndss-symposium.org

Consider, for instance, a client that offloads an encrypted database of (potentially sensitive) emails to an untrusted server. At a later point of time, the client might want to issue a query of the form "*retrieve all emails received from xyz@foobar.org* or "*retrieve all emails with the keyword "research" in the subject field*". Ideally, the client should be able to perform this task without revealing any sensitive information to the server, such as the sources and contents of the emails, the keywords underlying a given query, the distribution of keywords across emails, etc. Unfortunately, techniques such as fully homomorphic encryption [19], 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) [33], [20], [14], [32], [9], [8], [16], [36], [24], [29] is the study of provisioning symmetric-key encryption schemes with search capabilities. Consider again a client that offloads an encrypted database of emails to an untrusted server and later issues a query of the form "*retrieve all emails with the keyword "research" in the subject field*". The goal of SSE is to allow the client to perform this task without revealing any sensitive information to the server, such as the contents of emails, the keywords underlying a given query, the distribution of keywords across emails, etc.

Leakage Versus Efficiency. The most general notion of SSE with optimal security guarantees can be achieved using the work of Ostrovsky and Goldreich on Oblivious RAMs [21]. 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, *query pattern* (which queries correspond to the same keyword w) and the *access pattern* (the set of file identifiers matching a given query). Practical implementations of such schemes can be made extremely efficient and scalable using specially designed data structures.

Dynamic SSE. An important line of works (e.g., [11], [27], [26], [8], [5], [6], [15]) have studied *dynamic* SSE schemes that support updates on the database without the need to re-initialize the entire protocol. To formally address the additional privacy concerns that arise when supporting the update

operations, two new notions of security for SSE have been proposed in these works - (a) *forward privacy* (which requires that adding a new file f to a database should not reveal whether f contains keywords that have been previously searched for) and (b) *backward privacy* (which requires that searching for a keyword w should reveal no information about files containing w that have already been deleted from the database).

Forward private SSE was introduced by Chang and Mitzenmacher in [11], and has been subsequently studied in [35], [5], [18], [28], [6], [15], [34]. Forward privacy has received much attention in light of *file injection attacks* [7], [38], which are potentially devastating for SSE schemes that try to support updates without being forward private. The notion of backward privacy is comparatively more recent, and was first formalized by Bost *et al.* in [6]. Subsequently, Chamani *et al.* [10] and Sun *et al.* [37] proposed SSE schemes supporting single keyword search that are backward private under various leakage profiles.

However, existing dynamic SSE schemes, that satisfy *both* forward *and* backward privacy, support only single keyword search. As a result, despite their efficiency and security, these schemes are often severely limited in terms of the expressiveness of queries they support. Consider, for example, a client that can only specify a single keyword to search on, and receives all the documents containing this keyword. In real-life applications, such as querying large remotely stored email databases, a single keyword query would potentially return a large number of matching records/documents that the client would need to download and filter locally. For any SSE scheme to be truly practical, it should at least support conjunctive keyword search, i.e., given a set of keywords $(w_1, \ldots, w_n)$, it should be able to find and return the set of documents that contain *all* of these keywords.

Goals and Challenges. In this paper, we aim to design a dynamic SSE scheme with *both* forward *and* backward privacy, and with search complexity proportional to the number of documents containing the *least frequent term* in the conjunction. This is indeed the best possible search complexity achieved by plaintext information retrieval algorithms, as well as by conjunctive SSE schemes in the static setting [9], [29]. However, this is non-trivial to achieve in the dynamic SSE setting, where we need to additionally support updates and ensure forward and backward privacy. For instance, existing conjunctive SSE schemes in the static setting [9], [29] facilitate fast conjunctive searches by heavily pre-processing the dataset during setup. Such pre-processing at setup is impossible in the dynamic setting, where the dataset is updated on-the-fly.

Handling conjunctive searches also makes the analysis of leakage significantly more challenging. Existing definitions for forward and backward privacy [5], [6], [10], [37] assume leakage profiles that are tuned specifically towards single keyword search, and are insufficient to cover general conjunctive searches. For example, suppose that we design a dynamic SSE scheme that has the following leakage profile: given a conjunctive query over the keywords (w_1, w_2, w_3), it leaks to the server, in addition to the actual outcome of the query, the outcome of the sub-query (w_1, w_2). Note that this *partial* leakage is not meaningful when searching for a single keyword; so the aforementioned SSE scheme might well be secure according to forward/backward privacy definitions that cover *only* single keyword search. But for general conjunctive queries, such partial leakages could have devastating consequences [38].

A. Our Contributions

We develop the first dynamic SSE scheme supporting conjunctive keyword searches that is *both* forward *and* backward private. Our scheme is named Oblivious Dynamic Cross-Tags, or ODXT in short. The performance of ODXT scales to very large arbitrarily-structured databases, including both attribute-value and free-text databases.

Techniques Developed. The technical centerpiece of ODXT is a search protocol executed between the client and the server, where server takes as input a set of encrypted records corresponding to update operations on the database, while the client takes as input a conjunction of keywords and some secret state information. The outcome of this protocol is a filtered, significantly smaller set of encrypted records, which the client can then locally decrypt to compute the identifiers for documents containing all of the queried keywords.

A straightforward realization of this protocol, however, requires multiple rounds of communication between the client and the server, which does not satisfy our desired level of performance. In order to enable this search protocol with a *single round of communication*, we design a novel update mechanism based on *dynamic cross-tags* that *pre-computes* parts of the protocol messages, and stores these in encrypted form at the server. Then, during the actual search protocol, the client only sends across some auxiliary information that allows the server to unlock these pre-computed messages from the relevant update records, without any further interaction.

Differences with Static Cross-Tags. Our idea of pre-computing search protocol messages using cross-tags is inspired by conjunctive SSE schemes for static databases [9], [29]. However, applying this technique to the dynamic setting is not straightforward. In static SSE schemes, the pre-computation typically happens at setup, when the client has access to the entire database in the clear. Also, since the database is never updated, the pre-computed messages do not need to change with time. This is impossible to emulate in the dynamic setting, where the database is continuously updated. Finally, these schemes use specially designed data structures that are inherently static with no provisions for updates such as insertions/deletions.

This makes dynamic conjunctive SSE with appropriate performance and security guarantees non-trivial to achieve; in particular, prior attempts to do so have been found to be vulnerable to different classes of attacks such as leakage-abuse and file-injection attacks [7], [38].

Novelty of Our Approach. We introduce two novel techniques to tackle this issue that differ significantly from existing design-paradigms:

- A specialized data structure for "dynamic cross-tags" that can be efficiently updated and searched in tandem while ensuring both forward and backward-privacy.
- A round-reduction technique for conjunctive keyword searches that combines message pre-computation with

the update operations, and requires no pre-processing at setup.

At a high level, if an update operation (insertion/deletion) affects the outcome of some future search, we ensure that the corresponding message pre-computation for this search is also updated simultaneously. This combination of message pre-computation with normal update operations is done in a manner that: (a) leaks as little information as possible to the server, and (b) does not degrade the online efficiency of update and search operations.

Performance. Some of the performance benefits of ODXT are summarized below.

Fast Conjunctive Searches. Conjunctive keyword searches in ODXT entail only a single round of communication between the client and the server. The search complexity is independent of the total number of documents in the database. For a conjunctive query over a set of keywords $(w_1, \ldots, w_n)$, the search complexity of ODXT scales *linearly* with the number of update operations involving the *least frequent keyword* in the conjunction.

More specifically, the best possible search complexity for any conjunctive-SSE scheme is $O(n \cdot |\mathbf{DB}(w_1)|)$, where n is the number of keywords involved in the conjunction, w_1 is the least frequent of these keywords, and $|\mathbf{DB}(w_1)|$ is the number of files currently containing w_1. ODXT incurs slightly higher computational complexity, namely $O(n.|\mathsf{Upd}(w_1)|)$, where $|\mathsf{Upd}(w_1)|$ is the number update operations involving files containing w_1 (this is primarily a tradeoff for achieving both forward and backward privacy). Our experiments reveal that $|\mathsf{Upd}(w_1)|$ typically exceeds $|\mathbf{DB}(w_1)|$ by around 10%. In particular, any keyword that occurs in very few files is naturally expected to be involved in very few update operations.

In summary, ODXT achieves a search performance level "reasonably close" to the best possible search complexity achieved by plaintext information retrieval algorithms, as well as by conjunctive SSE schemes in the static setting [9], [29].

Fast Updates. Updates in ODXT are extremely fast and lightweight. Each update operation entails only a constant amount of computation at the client and the server, and a single message transmission from the client to the server. This matches closely the update efficiency of existing forward and backward private SSE schemes for single keyword search [6], [10], [37].

Efficient Storage. The server storage requirements for ODXT scale *linearly* with the number of update operations executed on the database until a given point of time, while the client is required to maintain a small amount of local storage that scales only *logarithmically* with the number of update operations executed on the database until a given point of time. This closely matches some of the most storage-efficient forward and backward private SSE schemes that support only single keyword search [6], [10], [37].

Security. We establish security by: (a) precisely enumerating the leakage profile for our scheme, including leakages from updates as well as leakages from conjunctive keyword searches, and then (b) by proving formally that this is indeed the *entire leakage* incurred by our scheme. Our formal proof of security follows the same simulation-based framework as existing forward and backward private SSE schemes for single keyword queries [6], [10], [37], and assumes an *adaptive* adversarial model. In this framework, we establish formally that a probabilistic polynomial-time simulation algorithm can simulate the view of the adversarial server (in a computationally indistinguishable manner) given access to only the leakage profile for our scheme.

Leakage Analysis. We also present a detailed analysis of the leakage profile incurred by our scheme, and compare it with the leakages incurred by existing forward and backward private SSE schemes supporting single keyword search, as well as existing conjunctive SSE schemes for static datasets. We broadly categorize the leakage from our scheme into two categories described below.

Update Leakages. These are leakages incurred during updates. The design of our scheme ensures that update operations reveal *nothing* to the adversary, including the nature of the update operation (insertion/deletion), as well as the document/keyword pair involved in the update operation.

Conjunctive Search Leakages. These are leakages incurred during conjunctive keyword searches. Examples of such leakages incurred by our scheme include the *access pattern*, the timestamps corresponding to updates involving the *least frequent term* in the conjunction, and the timestamps corresponding to updates involving other terms in the conjunction and the files containing the least frequent term. Some of these leakages are also incurred by existing forward and backward private in the single keyword search setting. Other leakages are very specific to the case of conjunctive queries, and we draw parallels with conjunctive SSE schemes in the static setting to justify their presence as a necessary performance trade-off.

Prototype Implementation. Finally, we present a prototype implementation of ODXT, and compare its search performances with the naïve adaptation of MITRA [10] to the conjunctive search setting, as well as IEX-2LEV and IEX-ZMF due to Kamara and Moataz [24]. The evaluations are carried out on 60.92GB-sized real world dataset obtained from Wikimedia downloads [17], consisting of 16 million documents, 43 million keywords and 100 million update operations.

B. Related Work

SSE for single keyword searches was first introduced by Song *et al.* in [33], and was subsequently equipped with formal security definitions by Goh in [20] and by Curtmola *et al.* in [14]. The literature on SSE that is relevant to this work can be broadly divided into two categories - dynamic SSE schemes that are forward and backward private but only support single keyword queries, and conjunctive SSE schemes that are either static or only forward private. We summarize them below.

Forward and Backward Private Dynamic SSE. The first SSE schemes to efficiently support updates [27], [26] were neither forward nor backward private. The notion of forward privacy was introduced formally in [11]. Since then, numerous works have proposed improved dynamic SSE schemes with forward privacy, albeit with support for single keyword

searches [35], [5], [18], [28], [6], [15], [34]. Backward privacy was introduced in [35], albeit without a formal security definition or construction. Bost *et al.* [6] introduced the first formal definitions of backward privacy for single keyword search, and proposed SSE constructions satisfying these notions. More efficient constructions of backward private SSE have been proposed subsequently in [37], [10].

To the best of our knowledge, all forward and backward private SSE constructions till date *only* support single keyword searches. In particular, they *do not support* conjunctive keyword searches, which is the goal of this paper.

Conjunctive SSE. A completely disjoint set of works have attempted to design SSE schemes that support expressive queries such as conjunctions, disjunctions and general Boolean formulae over keywords. The seminal work of Cash *et al.* [9] and a subsequent work of Lai *et al.* [29] enable efficient conjunctive keyword searches, albeit on static datasets with no provisions for updates. The work of Kamara and Moataz [24] enables conjunctive keyword searches over dynamic databases, but is *only* forward private.

In this work, we address the open question of designing an SSE scheme for conjunctive keyword searches over dynamic databases while *simultaneously* achieving both forward and backward privacy.

II. Preliminaries

In this section we introduce the notations that are used in the rest of the paper. We refer the reader to the full version of the paper [31] for additional cryptographic background and background material on dynamic SSE.

Notations. We write $x \xleftarrow{R} \chi$ to represent that an element x is sampled uniformly at random from a set/distribution $\mathcal{X}$. The output x of a deterministic algorithm $\mathcal{A}$ is denoted by $x = \mathcal{A}$ and the output x' of a randomized algorithm $\mathcal{A}'$ is denoted by $x' \leftarrow \mathcal{A}'$. For $a \in \mathbb{N}$ such that $a \geq 1$, we denote by $[a]$ the set of integers lying between 1 and a (both inclusive). We refer to $\lambda \in \mathbb{N}$ as the security parameter, and denote by $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ any generic (unspecified) polynomial function and negligible function in λ, respectively. [1]

Databases. Let $\Delta = \{w_1, \ldots, w_K\}$ be a dictionary of keywords, and let $\mathcal{F} = \{f_1, \ldots, f_D\}$ be a collection of files, such that each f_i is associated with a unique identifier id_i and contains keywords from Δ. We denote by **DB** a database of identifier-keyword pairs, such that a given pair $(\mathsf{id}, w) \in \mathbf{DB}$ if and only if the file with identifier id contains the keyword w. We denote by $\mathcal{W} \subseteq \Delta$ the set of all keywords that appear at least once in **DB**, and by $\mathbf{DB}(w)$ the set of all identifiers corresponding to files containing w. We denote by $|\mathcal{W}|$ the number of distinct keywords in **DB**, by $|\mathbf{DB}|$ the number of distinct identifier-keyword pairs in **DB**, by $|\mathbf{DB}(w)|$ the number of files containing the keyword w, and by $|\mathsf{Upd}(w)|$ the number of update operations involving the keyword w.

Conjunctive Queries. We represent a conjunctive query over n distinct keywords $w_1, \ldots, w_n$ as $q = (w_1 \wedge w_2 \wedge \ldots \wedge w_n)$ and define the set $\mathbf{DB}(q)$ as $\mathbf{DB}(q) = \cap_{i=1}^{n} \mathbf{DB}(w_i)$. Depending on the context, the keyword w_1 is assumed to have either the least frequency of occurrence or to have the least frequency of updates among all keywords in the conjunction q.

Dynamic SSE. A dynamic searchable symmetric encryption (SSE) scheme consists of a polynomial-time algorithm Setup executed by the client, and protocols Search and Update executed jointly by the client and the server:

- $\textsc{Setup}(\lambda)$: A probabilistic algorithm that takes the security parameter λ. It outputs the tuple $(\mathsf{sk}, \mathsf{st}, \mathbf{EDB})$, where sk is the client's secret-key, st is the client's internal state, and **EDB** is an *empty* encrypted database.
- $\textsc{Update}(\mathsf{sk}, \mathsf{st}, \mathsf{op}, (\mathsf{id}, w); \mathbf{EDB})$: A client-server protocol, where the client takes as input the secret-key sk, its state st, an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ and an identifier-keyword pair (id, w), while the server takes as input the encrypted database **EDB**. The protocol outputs a modified client state st' and a modified encrypted database $\mathbf{EDB}'$ so as to reflect the outcome of the addition/deletion operation.
- $\textsc{Search}(\mathsf{sk}, \mathsf{st}, q; \mathbf{EDB})$: A client-server protocol, where the client takes as input the secret-key sk, its state st and a query q, while the server takes as input the encrypted database **EDB**. At the end of the protocol, the client outputs $\mathbf{DB}(q)$.

In the above, we adopted the definition of dynamic SSE used by Chamani *et al.* [10]. There exist other definitions of dynamic SSE in the literature [28], [15] where the Update operation takes an entire file for addition/deletion, which is functionally equivalent to executing multiple addition/deletion operations on the relevant identifier/keyword pairs in our framework. Finally, we make the implicit assumption that upon obtaining the set of file identifiers corresponding to a query, the client performs an additional interaction with the server to actually retrieve the files with these identifiers.

Correctness. A dynamic SSE is said to be correct if for every database **DB** and for every query q, the Search protocol outputs $\mathbf{DB}(q)$ with all but negligible probability.

Security. We refer the reader to the full version of the paper [31] for the formal security definition of a dynamic SSE scheme.

III. Dynamic Conjunctive SSE Schemes

A. A Naïve Solution

To motivate our solutions, we begin with a straightforward extension of the dynamic SSE scheme Mitra introduced by Chamani et al. [10] from single keyword queries to conjunctive queries.[2] The idea is as follows: on input of a conjunctive query $q = (w_1 \wedge \ldots \wedge w_n)$, the client and the server run the original Mitra search protocol in parallel for each keyword w_i. At the end of the search protocol, the client receives a list

[1] 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.

[2] We choose Mitra because it has the best update and search performances in practice among existing forward and backward private SSE scheme. However, conceptually, the extension works for all forward and backward private SSE schemes supporting single keyword search.

Client

1) Sample a uniformly random key K_T for PRF F
2) Initialize UpdateCnt, TSet to empty maps
3) Set $\mathsf{sk} = K_T$ and $\mathsf{st} = \mathsf{UpdateCnt}$
4) Set **EDB** $= \mathsf{TSet}$
5) Send **EDB** to the server

Figure 1: $\text{MITRA}_{\text{CONJ}}$. SETUP ($\lambda$)

of encrypted file identifiers corresponding to each keyword, decrypts each such list, and retains only the file identifiers in the intersection of all the lists.

We refer to this naïve adaptation of MITRA for conjunctive queries as $\text{MITRA}_{\text{CONJ}}$. The corresponding setup, update and search algorithms are described in Figures 1, 2 and 3, respectively. Below, we provide a brief technical overview of how $\text{MITRA}_{\text{CONJ}}$ handles conjunctive queries. For more details on the original MITRA scheme, the reader may refer to [10].

Construction Overview. The construction of $\text{MITRA}_{\text{CONJ}}$ is based on a key-value dictionary called a TSet designed as follows: for each keyword w, the TSet dictionary stores encrypted transcripts corresponding to each operation involving w. The keys for TSet (which are addresses in the dictionary storing encrypted values) are generated using a PRF.

During an update operation of the form $[\mathsf{op}(\mathsf{id}, w)]$, the client generates the appropriate key-value pair for the TSet dictionary, and sends it over to the server. The server updates the dictionaries accordingly. Under the assumption that file identifiers are never repeated[3], the use of PRFs ensures that these key-value pairs reveal no information to the server about the underlying operation op, the identifier id or the keyword w. Since updates are leakage-free, forward privacy follows immediately.

Finally, let $q = (w_1 \wedge w_2 \wedge \ldots \wedge w_n)$ be a conjunctive query issued by the client. For each keyword w_i (in parallel), the client recovers $\mathbf{DB}(w_i)$ via the following steps. The client efficiently generates the appropriate keys for the TSet dictionary corresponding to each operation involving the keyword w_i, and sends these across to the server. The server retrieves the encrypted transcripts corresponding to each operation involving w_i and sends these back to the client. Upon receiving the encrypted transcripts, the client decrypts them to recover each update operation involving w_i. Given this information, constructing $\mathbf{DB}(w_i)$ is straightforward. Eventually, the client computes $\mathbf{DB}(q) = \cap_{i=1}^{n} \mathbf{DB}(w_i)$.

Search Performance. It is straightforward to observe that the computational and communication complexity of this search protocol is proportional to $\sum_{i=1}^{n} |\mathsf{Upd}(w_i)|$, which is at least as large as $\sum_{i=1}^{n} |\mathbf{DB}(w_i)|$. This may be reasonable in practice if each keyword w_i is low-frequency, but is definitely rather poor if one or more keywords have very high-frequency of occurrence.

[3] This assumption is made in several existing forward and backward private SSE schemes for single keyword search, most notably in the constructions of Bost *et al.* [6] and Chamani *et al.* [10], including the original MITRA scheme.

Client

1) Parse $\mathsf{sk} = K_T$ and $\mathsf{st} = \mathsf{UpdateCnt}$
2) **If** $\mathsf{UpdateCnt}[w]$ is NULL **then** set $\mathsf{UpdateCnt}[w] = 0$
3) Set $\mathsf{UpdateCnt}[w] = \mathsf{UpdateCnt}[w] + 1$
4) Set $\mathsf{addr} = F(K_T, w||\mathsf{UpdateCnt}[w]||0)$
5) Set $\mathsf{val} = (\mathsf{id}||\mathsf{op}) \oplus F(K_T, w||\mathsf{UpdateCnt}[w]||1)$
6) Send $(\mathsf{addr}, \mathsf{val})$ to the server

Server

1) Parse **EDB** $= \mathsf{TSet}$
2) Set $\mathsf{TSet}[\mathsf{addr}] = \mathsf{val}$

Figure 2: $\text{MITRA}_{\text{CONJ}}$. UPDATE $(\mathsf{sk}, \mathsf{st}, \mathsf{op}, (\mathsf{id}, w); \mathbf{EDB})$

Client

1) Parse $\mathsf{sk} = K_T$ and $\mathsf{st} = \mathsf{UpdateCnt}$
2) Initialize $\mathsf{tokenList}_1, \ldots, \mathsf{tokenList}_n$ to empty lists
3) **For** $i = 1$ **to** n:
   a) **For** $j = 1$ **to** $\mathsf{UpdateCnt}[w_i]$:
      i) Set $\mathsf{addr}_{i,j} = F(K_T, w_i||j||0)$
      ii) Set $\mathsf{tokenList}_i = \mathsf{tokenList}_i \cup \{\mathsf{addr}_{i,j}\}$
   b) **End For**
4) **End For**
5) Send $\mathsf{tokenList}_1, \ldots, \mathsf{tokenList}_n$ to the server

Server

1) Parse **EDB** $= \mathsf{TSet}$
2) Initialize $\mathsf{EOpList}_1, \ldots, \mathsf{EOpList}_n$ to empty lists
3) **For** $i = 1$ **to** n:
   a) **For** $j = 1$ **to** $\mathsf{tokenList}_i.\mathsf{size}$:
      i) Set $\mathsf{val}_{i,j} = \mathsf{TSet}[\mathsf{tokenList}_i[j]]$
      ii) Set $\mathsf{EOpList}_i = \mathsf{EOpList}_i \cup \{\mathsf{val}_{i,j}\}$
   b) **End For**
4) **End For**
5) Send $\mathsf{EOpList}_1, \ldots, \mathsf{EOpList}_n$ to the client

Client: Final Output Computation

1) Initialize $\mathsf{IdList}_1, \ldots, \mathsf{IdList}_n$ to empty lists
2) **For** $i = 1$ **to** n:
   a) **For** $j = 1$ **to** $\mathsf{UpdateCnt}[w_i]$:
      i) Set:
      $$(\mathsf{id}_{i,j}||\mathsf{op}_{i,j}) = \mathsf{EOpList}_i[j] \oplus F(K_T, w_i||j||1)$$
      ii) **If** $\mathsf{op}_{i,j}$ is add **then** set $\mathsf{IdList}_i = \mathsf{IdList}_i \cup \{\mathsf{id}_{i,j}\}$
      iii) **Else** set $\mathsf{IdList}_i = \mathsf{IdList}_i \setminus \{\mathsf{id}_{i,j}\}$
   b) **End For**
3) **End For**
4) Output $\mathsf{IdList} = \cap_{i=1}^{n} \mathsf{IdList}_i$

Figure 3: $\text{MITRA}_{\text{CONJ}}$. SEARCH $(\mathsf{sk}, \mathsf{st}, q = (w_1 \wedge \ldots \wedge w_n); \mathbf{EDB})$

Leakage. Although this scheme inherits many of the forward and backward privacy properties of the original MITRA scheme, it incurs an additional undesirable leakage: a search operation over keywords $w_1, \ldots, w_n$ allows the server to learn $|\mathsf{Upd}(w_i)|$ (i.e., the total number of update operations) for each keyword w_i, including those involving files that are not relevant to the query, and the corresponding timestamp associated with each such update operation.

Our goal is to reduce both the computational overheads as well as the leakages in the protocol by tying these to only the less frequent keywords in the queried conjunction.

B. Basic Dynamic Cross-Tags

To achieve the above goal, we introduce the idea of "dynamic cross-tags". For ease of understanding, we exemplify the idea via a simplified protocol, called Basic Dynamic Cross-Tags, or BDXT in short. The corresponding algorithms for setup, updates and search are described in Figures 4, 5 and 6 , respectively. The main changes from $\text{MITRA}_{\text{CONJ}}$ are highlighted in red.

Assume that, given a conjunctive query $q = (w_1 \wedge \ldots \wedge w_n)$, the client can choose the keyword with the least frequency of occurrence (at the cost of small additional storage). Assume without loss of generality that this keyword is w_1. We will refer to w_1 as the *s-term* (where s stands for "special") and to each of the remaining keywords $w_2, \ldots, w_n$ as a *x-term* (where x stands for "cross").

Handling the s-Term. In our simplified protocol presented below, the client still runs an instance of the MITRA search protocol, albeit *only* for the s-term w_1, following which the client is able to retrieve the set of all identifiers corresponding to files currently containing w_1. In the process, the computational overheads incurred by the client and the server are both proportional to $\mathbf{DB}(w_1)$, and the server only learns $|\mathbf{DB}(w_1)|$ (assuming no padding for now).

At this point, an obvious solution is as follows: the client downloads all the files containing w_1, parses them locally and retains only those files that contain all the other keywords $w_2, \ldots, w_n$. This is extremely inefficient from a performance point of view, since it requires downloading and parsing many more files than actually necessary. In order to handle this more efficiently, we introduce the idea of "dynamic cross-tags" below.

Dynamic Cross-Tags. Concretely, in addition to the TSet dictionary in the previous scheme, we use an additional dictionary called the XSet that has *a pair of* designated addresses for each possible identifier-keyword pair (id, w). At any given time, this address pair is populated with one of the following value pairs:

- $(\perp, \perp) : (\mathsf{id}, w)$ was neither inserted nor deleted
- $(1, \perp) : (\mathsf{id}, w)$ was inserted but not yet deleted
- $(1, 1) : (\mathsf{id}, w)$ was inserted and later deleted

where $\perp$ denotes the corresponding address is empty. The keys pointing to these addresses are referred to as "dynamic cross-tags", and represent a major technical contribution of this work. Unlike the "cross-tags" in the scheme of Cash *et al.* [9] which can only determine the presence/absence of any identifier-keyword pair in a static dataset, the keys for our XSet dictionary can determine the presence/absence of any identifier-keyword pair in a dynamic dataset across any number of update operations.

These dynamic cross-tags are generated using PRFs, so that they may be efficiently reproduced by the client during update/search queries. More concretely, for an identifier-keyword pair (id_j, w_i), the corresponding "insertion-cross-tag" and "deletion-cross-tag" are generated as:

$$\mathsf{xtag}_{i,j,\mathsf{add}} = F(K_X, w_i || \mathsf{id}_j || \mathsf{add}) \ , \ \mathsf{xtag}_{i,j,\mathsf{del}} = F(K_X, w_i || \mathsf{id}_j || \mathsf{del}).$$

This is illustrated in Figure 5.

Client

1) Sample a uniformly random key K_T, K_X for PRF F
2) Initialize UpdateCnt, DBCnt, TSet, XSet to empty maps
3) Set $\mathsf{sk} = (K_T, K_X)$ and $\mathsf{st} = (\mathsf{UpdateCnt}, \mathsf{DBCnt})$
4) Set $\mathbf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$
5) Send **EDB** to the server

Figure 4: BDXT. SETUP (λ)

Client

1) Parse $\mathsf{sk} = (K_T, K_X)$ and $\mathsf{st} = (\mathsf{UpdateCnt}, \mathsf{DBCnt})$
2) **If** $\mathsf{UpdateCnt}[w]$ is NULL **then** set:

$$\mathsf{UpdateCnt}[w] = \mathsf{DBCnt}[w] = 0$$

3) Set $\mathsf{UpdateCnt}[w] = \mathsf{UpdateCnt}[w] + 1$
4) **If** op is add **then** set $\mathsf{DBCnt}[w] = \mathsf{DBCnt}[w] + 1$
5) **Else** set $\mathsf{DBCnt}[w] = \mathsf{DBCnt}[w] - 1$
6) Set $\mathsf{addr} = F(K_T, w||\mathsf{UpdateCnt}[w]||0)$
7) Set $\mathsf{val} = (\mathsf{id}||\mathsf{op}) \oplus F(K_T, w||\mathsf{UpdateCnt}[w]||1)$
8) Set $\mathsf{xtag} = F(K_X, w||\mathsf{id}||\mathsf{op})$
9) Send $(\mathsf{addr}, \mathsf{val}, \mathsf{xtag})$ to the server

Server

1) Parse $\mathbf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$
2) Set $\mathsf{TSet}[\mathsf{addr}] = \mathsf{val}$
3) Set $\mathsf{XSet}[\mathsf{xtag}] = 1$

Figure 5: BDXT. UPDATE $(\mathsf{sk}, \mathsf{st}, \mathsf{op}, (\mathsf{id}, w); \mathbf{EDB})$

Handling Updates. The update procedure for BDXT is described in Figure 5. The TSet dictionary is updated as in the previous scheme $\text{MITRA}_{\text{CONJ}}$, and hence incurs no leakages. The XSet dictionary is updated as follows: when an identifier-keyword pair (id, w) is inserted, the entry at the "insertion cross-tag" corresponding to (id, w) is updated to 1. At a later time, when (id, w) is deleted, the entry at the "deletion-cross-tag" corresponding to (id, w) is updated to 1.

Differences with Static Cross-Tags. A key difference in our approach as compared to conjunctive SSE schemes for static databases [9], [29] is that our cross-tags are computed on-the-fly with every update operation, and not at setup. In the works of Cash *et al.* [9] and Lai *et al.* [9], the presence or absence of a cross tag in the XSet simply indicated whether a given file contains a certain keyword or not. By involving the operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ in the generation of the cross-tag, we have extended its semantic meaning to now indicate whether a certain operation (either addition or deletion) involving a given keyword-file pair has occurred or not. As a result, the XSet data structure, which was an inherently static data structure in the previous works, is now transformed into a dynamic data structure that can be updated without any additional pre-computation at setup. We managed to do this while maintaining forward privacy (because a cross-tag does not reveal any information about the underlying operation, file identifier or keyword), which is crucial for achieving resistance against leakage-abuse attacks [7] and file-injection attacks [38].

In addition, as we demonstrate subsequently, our dynamic cross-tags are *both* forward *and* backward private, in the sense that they also incur minimal leakages during conjunctive searches. In particular, our technique of treating additions and

Client: Round 1

1) Parse $\mathsf{sk} = (K_T, K_X)$ and $\mathsf{st} = (\mathsf{UpdateCnt}, \mathsf{DBCnt})$
2) Use DBCnt to identify the least frequent keyword (assumed to be w_1 w.l.o.g)
3) Initialize stokenList to an empty list
4) **For** $j = 1$ **to** $|\mathsf{Upd}(w_1)|$:
   a) Set $\mathsf{saddr}_j = F(K_T, w_1||j||0)$
   b) Set stokenList = stokenList $\cup \{\mathsf{saddr}_j\}$
5) **End For**
6) Send stokenList to the server

Server: Round 1

1) Parse **EDB** $= (\mathsf{TSet}, \mathsf{XSet})$
2) Initialize sEOpList to an empty list
3) **For** $j = 1$ **to** stokenList.size:
   a) Set $\mathsf{sval}_j = \mathsf{TSet}[\mathsf{stokenList}[j]]$
   b) Set sEOpList = sEOpList $\cup \{\mathsf{sval}_j\}$
4) **End For**
5) Send sEOpList to the client

Client: Round 2

1) Initialize sIdList to an empty list
2) **For** $j = 1$ **to** $|\mathsf{Upd}(w_1)|$:
   a) Set $(\mathsf{id}_j||\mathsf{op}_j) = \mathsf{sEOpList}[j] \oplus F(K_T, w_1||j||1)$
   b) **If** op_j is add **then** set sIdList = sIdList $\cup \{\mathsf{id}_j\}$
   c) **Else** set sIdList = sIdList $\setminus \{\mathsf{id}_j\}$
3) **End For**
4) Let $m =$ sIdList.size (=$|\mathbf{DB}(w_1)|$).
5) Initialize $\mathsf{xtagList}_1, \ldots, \mathsf{xtagList}_m$ to empty lists
6) **For** $j = 1$ **to** m:
   a) Let $\mathsf{id}_j = \mathsf{sIdList}[j]$
   b) **For** $i = 2$ **to** n:
      i) Set $\mathsf{xtag}_{i,j,\mathsf{add}} = F(K_X, w_i||\mathsf{id}_j||\mathsf{add})$
      ii) Set $\mathsf{xtag}_{i,j,\mathsf{del}} = F(K_X, w_i||\mathsf{id}_j||\mathsf{del})$
      iii) Set $\mathsf{xtagList}_j = \mathsf{xtagList}_j \cup \{(\mathsf{xtag}_{i,j,\mathsf{add}}, \mathsf{xtag}_{i,j,\mathsf{del}})\}$
   c) Randomly permute the tuple-entries of $\mathsf{xtagList}_j$
7) **End For**
8) Send $(\mathsf{xtagList}_1, \ldots, \mathsf{xtagList}_m)$ to the server

Server: Round 2

1) **For** $j = 1$ **to** m:
   a) Set $b_j = 1$
   b) **For** $i = 2$ **to** n:
      i) Set $(\mathsf{xtag}_{i,j,\mathsf{add}}, \mathsf{xtag}_{i,j,\mathsf{del}}) = \mathsf{xtagList}_j[i]$
      ii) **If** $\mathsf{XSet}[\mathsf{xtag}_{i,j,\mathsf{add}}] = \perp$, **then** set $b_j = 0$
      iii) **Else If** $\mathsf{XSet}[\mathsf{xtag}_{i,j,\mathsf{del}}] = 1$, **then** set $b_j = 0$
   c) **End For**
2) **End For**
3) Send $(b_1, \ldots, b_m)$ to the client

Client: Final Output Computation

1) Initialize IdList to an empty list
2) **For** $j = 1$ **to** m:
   a) Let $\mathsf{id}_j = \mathsf{sIdList}[j]$
   b) If $b_j = 1$, then set IdList = IdList $\cup \{\mathsf{id}_j\}$
3) **End For**
4) Output IdList

Figure 6: BDXT. SEARCH $(\mathsf{sk}, \mathsf{st}, q = (w_1 \wedge \ldots \wedge w_n); \mathbf{EDB})$

deletions in a symmetric manner by generating cross-tags for them using the same PRF operation ensures that the adversary also cannot infer additional information about the deletion history of keywords (it is computationally indistinguishable from the insertion history), which is the primary requirement for backward privacy. Achieving simultaneously forward *and* backward private dynamic cross-tags constitutes the key technical innovation of our work and has not, to our knowledge, been achieved by prior works.

Handling Conjunctive Searches. The conjunctive search procedure for BDXT is described in Figure 6. Let $q = (w_1 \wedge w_2 \wedge \ldots \wedge w_n)$ be a conjunctive query issued by the client, and let w_1 be the keyword with the least frequency. In our simplified protocol, the search operation involves two rounds of communication between the client and the server.

Round-1 allows the client to recover $\mathbf{DB}(w_1)$ as mentioned above. More concretely, the client first efficiently generates all relevant addresses in the TSet related to w_1 and sends them across to the server. The server then retrieves the encrypted $(\mathsf{id}, \mathsf{op})$ pairs and transmits them back to the client. Finally, the client locally decrypts and recovers $\mathbf{DB}(w_1)$. This is very similar to the search algorithm in MITRA$_{\text{CONJ}}$.

Round-2 is based on the following observation: at a given point of time, an identifier-keyword pair $(\mathsf{id}_j, w_i) \in \mathbf{DB}$ iff the following conditions hold simultaneously: (a) the "insertion-cross-tag" corresponding to (id_j, w_i) is currently set to 1 (meaning that (id_j, w_i) has been inserted), *and* (b) the "deletion-cross-tag" corresponding to (id_j, w_i) is currently set to $\perp$ (meaning that (id_j, w_i) is not yet deleted).

Based on this observation, it is natural to execute **Round-2** of the conjunctive search via the following steps:

1) For each identifier $\mathsf{id}_j \in \mathbf{DB}(w_1)$, the client efficiently computes the cross-tag-pairs corresponding to $(\mathsf{id}_j, w_2), \ldots, (\mathsf{id}_j, w_n)$, and sends these $(n-1)$ cross-tag-pairs across to the server (in randomly permuted order).
2) For each $j \in |\mathbf{DB}(w_1)|$, the server receives a set of $(n-1)$ cross-tag-pairs from the client and retrieves the corresponding XSet entries. If for each pair, the first entry is 1 and second entry is $\perp$, the server returns $b_j = 1$, otherwise it returns $b_j = 0$.
3) For each $\mathsf{id}_j \in \mathbf{DB}(w_1)$, if the corresponding bit b_j received from the server is 1, the client includes the identifier id_j in the final list of identifiers to be output. Otherwise, it discards the identifier id_j.

Correctness of the search protocol follows immediately from the aforementioned observation.

Implementing XSet**.** The XSet dictionary is represented equivalently using a set $\mathcal{S}_{\mathsf{XSet}}$ that is history-independent (i.e., it is independent of the order in which the elements of the set were inserted), and supports: (a) efficient element insertion and (b) efficient membership test for a random element. For a dynamic cross-tag $\mathsf{xtag}_{i,j,\mathsf{op}}$ corresponding to an identifier-keyword pair (id_j, w_i) and an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, we interpret its corresponding value in the XSet dictionary as:

$$\mathsf{XSet}[\mathsf{xtag}_{i,j,\mathsf{op}}] = \begin{cases} 1 & \text{if } \mathsf{xtag}_{i,j,\mathsf{op}} \in \mathcal{S}_{\mathsf{XSet}} \\ \perp & \text{otherwise} \end{cases}$$

During an update operation, setting a XSet entry to 1 can be realized by simply adding the corresponding cross-tag to the set $\mathcal{S}_{\mathsf{XSet}}$. As long as $\mathcal{S}_{\mathsf{XSet}}$ supports efficient element insertion, an update operation can thus be realized efficiently. Similarly, as long as $\mathcal{S}_{\mathsf{XSet}}$ supports efficient membership testing, the XSet dictionary can be efficiently looked up by the server during conjunctive searches.

Server Storage. The server stores the dictionaries TSet and XSet. Note that during setup, the TSet and XSet dictionaries are both initialized to empty. After N updates, the storage requirement at the server grows linearly as $O(N\lambda)$, since each update operation adds a $O(\lambda)$-sized entry of the form $(\mathsf{addr}, \mathsf{val})$ to TSet and a $O(\lambda)$-sized cross-tag entry of the form $(\mathsf{xtag}, 1)$ to XSet. In other words, the storage requirement at the server grows *linearly* with the number of update operations on the dataset.

Client Storage. The client locally stores the arrays $\mathsf{UpdateCnt}$ and DBCnt. Note that during setup, both arrays are initialized to empty. After N updates, the storage requirement at the client grows as $O(|\mathcal{W}| \cdot \log N)$, $|\mathcal{W}|$ is the size of the keyword dictionary, which is typically upper-bounded by some large pre-defined constant. In other words, the storage requirement at the client grows *logarithmically* with the number of update operations on the dataset.

Search Performance. The computational overhead at both the client and the server scales with $(|\mathsf{Upd}(w_1)| + (n-1) \cdot |\mathbf{DB}(w_1)|)$. This is clearly a significant improvement over the naïve adaptation over MITRA whenever there is a query term in the conjunction with relatively small frequency of occurrence. The communication overhead also scales with $(|\mathsf{Upd}(w_1)| + (n-1) \cdot |\mathbf{DB}(w_1)|)$, which is again a significant improvement over the naïve adaptation over MITRA whenever $\mathbf{DB}(w_1)$ is small. In particular, this matches our original goal of reducing the computational and communication overheads by tying these to the s-term w_1 that has the lowest frequency of occurrence.

An undesirable feature of BDXT from the point of view of search performance is the extra round of communication with consequent latency. For some applications, low latency might be a more crucial requirement and having a single round of communication during searches might be preferable, even if at the cost of additional computation at the client and/or server. Having multiple rounds of interaction during searches also limits the applicability of BDXT to some settings, such as the multi-client SSE setting. We expand on this subsequently.

Leakage. In terms of leakage, BDXT again improves substantially upon the naïve adaptation of MITRA by tying the leakage from conjunctive searches to the s-term w_1 that has the least frequency of occurrence. Recall that in $\text{MITRA}_{\text{CONJ}}$, a search operation allows the server to learn partial information about every update operation involving every keyword in the conjunction. On the other hand, in BDXT, for each x-term in $\{w_2, \ldots, w_n\}$, the information gained by the adversary is only restricted to update operations involving files in $\mathbf{DB}(w_1)$. To see this, observe that if a file with identifier id contains some x-term (say, w_2) but does not contain the s-term w_1, then in BDXT, the server does not receive any cross-tag corresponding to id, and hence learns no information about the pair (id, w_2).

However, BDXT still leaks more information than desirable. To begin with, BDXT allows the server to learn the frequency of the s-term, i.e., $|\mathbf{DB}(w_1)|$, in addition to the number of update operations involving the s-term, i.e., $|\mathsf{Upd}(w_1)|$. This immediately leaks the exact number of insertion and deletion operations involving w_1. Note that the naïve adaptation of MITRA to the conjunctive setting does not suffer from this leakage, as it only reveals $|\mathsf{Upd}(w_1)|$ to the server.

BDXT also allows the server to learn cross-tag pairs in the XSet dictionary that correspond to the *same* identifier-keyword pair, as well as the update history for this pair. Although the server cannot immediately identify *which keyword* among the $\mathbf{x}$-terms $w_2, \ldots, w_n$ a given cross-tag pair corresponds to (since the cross-tag pairs are uniformly randomly permuted for each file identifier in $\mathbf{DB}(w_1)$), it can test each cross-tag pair for membership in the XSet dictionary to learn the exact number of keywords among $w_2, \ldots, w_n$ that each file in $\mathbf{DB}(w_1)$ contains.

We present in the next subsection an improved version of BDXT that achieves significantly smaller leakage; hence, we avoid a formal analysis of the leakage of BDXT.

C. Oblivious Dynamic Cross-Tags

We address the drawbacks of BDXT with respect to both search performance and leakage by presenting an alternative realization of dynamic cross-tags called Oblivious Dynamic Cross-Tags, or ODXT in short. The corresponding algorithms for setup, updates and search are described in Figures 7, 8 and 9 , respectively. The main changes from BDXT are highlighted in red.

The key technical difference between ODXT and BDXT is that ODXT uses an oblivious shared computation between the client and the server to allow conjunctive searches with a single round of communication. To enable this oblivious shared computation, we resort to using *blinded exponentiations* (as in the Diffie-Hellman based oblivious PRF) in a cyclic group of prime order. ODXT also improves upon BDXT in terms of search privacy by reducing the information leakage to the server during conjunctive searches.

The Idea. In order to elucidate the core idea behind ODXT, we focus on why our simpler scheme, namely BDXT, requires two rounds of communication between the server and the client. Note that in the first round, the client executes a single keyword search on the s-term to recover $\mathbf{DB}(w_1)$. Consequently, in the second round, it generates a pair of cross-tags $(\mathsf{xtag}_{i,j,\mathsf{add}}, \mathsf{xtag}_{i,j,\mathsf{del}})$ for each keyword $w_i \in \{w_2, \ldots, w_n\}$ and each document identifier $\mathsf{id}_j \in \mathbf{DB}(w_1)$ recovered in the first round. If the client could allow the server to compute these cross-tags *without explicitly recovering* $\mathbf{DB}(w_1)$, the additional round communication could be avoided.

Our goal is to enable an oblivious evaluation of the cross-tag pair without explicitly recovering $\mathbf{DB}(w_1)$*, thereby avoiding an additional round of interaction between the client and the server.*

Change Cross-Tags in XSet. The first step in realizing this goal is to change the manner in which the cross-tags are generated. For a keyword w_i, a document identifier id_j and an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, the client now generates the corresponding cross-tag $\mathsf{xtag}_{i,j,\mathsf{op}}$ as

$$\mathsf{xtag}_{i,j,\mathsf{op}} = g^{F_p(K_X, w_i) \cdot F_p(K_Y, \mathsf{id}_j || \mathsf{op})},$$

where g is a generator for a cyclic group $\mathcal{G}$ of prime order p, F_p is a PRF with range Z_p^*, and K_X and K_Y are uniformly sampled keys for the PRF F_p.

Client

1) Sample a uniformly random key K_T for PRF F
2) Sample uniformly random keys K_X, K_Y, K_Z for PRF F_p
3) Initialize UpdateCnt, TSet, XSet to empty maps
4) Set $\mathsf{sk} = (K_T, K_X, K_Y, K_Z)$ and $\mathsf{st} = \mathsf{UpdateCnt}$
5) Set **EDB** $= (\mathsf{TSet}, \mathsf{XSet})$
6) Send **EDB** to the server

Figure 7: ODXT. SETUP (λ)

Note that conceptually, the xtag is split into two parts, one pertaining to w_i and the other pertaining to the pair $(\mathsf{id}_j, \mathsf{op})$, which are combined multiplicatively in the exponent of g. This is the key change from how the xtag was generated in BDXT (in BDXT, these two parts were combined into a single PRF evaluation). As we shall see, this is crucial to enabling the oblivious computation.

Note: The tag calculation mechanism works even when a given document is being updated with the same keyword(s) multiple times. As stated earlier in footnote 3, we assume that update operations involving the same file identifier are never repeated. In particular, when an existing file is to be updated, it is deleted and re-inserted (in modified form) under a fresh file identifier. This assumption is made in several existing forward and backward private SSE schemes for single keyword search, most notably in the constructions of Bost *et al.* [6] and Chamani *et al.* [10], including the original MITRA scheme.

Dynamic Blinding Factors in TSet. The client also computes and stores in the TSet dictionary a *dynamic blinding element* corresponding to each update operation. For example, let $(\mathsf{op}, (\mathsf{id}_j, w_i))$ be the cnt^{th} update operation involving the keyword w_i (the client can keep track of this count for each keyword using the UpdateCnt data structure). In the TSet address corresponding to this update operation, the client additionally stores the following blinding element:

$$\alpha_{i,j,\mathsf{op}} = F_p(K_Y, \mathsf{id}_j||\mathsf{op}) \cdot (F_p(K_Z, w_i||\mathsf{cnt}))^{-1},$$

where g, F_p and K_Y are as defined before, and K_Z is again a uniformly sampled key for the PRF F_p.

Note again that conceptually, the blinding factor α is also split into two parts, one pertaining to the keyword-count pair (w_i, cnt) and the other pertaining to the pair $(\mathsf{id}_j, \mathsf{op})$, which are combined multiplicatively in Z_p^*. Also note that the part pertaining to the pair $(\mathsf{id}_j, \mathsf{op})$ is the same in both the xtag and the blinding factor α. This is an intentional design choice. Looking ahead, during a search operation, the server will be provided with a "search token" that, when "obliviously" combined with the blinding term α, will give rise to an expression that matches the corresponding xtag. The presence or absence of this xtag in the XSet will then determine the outcome of the search. We present the details of this oblivious combination mechanism next.

Differences with Static Cross-Tags and Static Blinding Factors. Once again, unlike previous works [9], [29], our cross-tags are computed on-the-fly with every update operation, and not at setup. In the OXT scheme of *et al.* [9] and the HXT

Client

1) Parse $\mathsf{sk} = (K_T, K_X, K_Y, K_Z)$ and $\mathsf{st} = \mathsf{UpdateCnt}$
2) **If** $\mathsf{UpdateCnt}[w]$ is NULL **then** set $\mathsf{UpdateCnt}[w] = 0$
3) Set $\mathsf{UpdateCnt}[w] = \mathsf{UpdateCnt}[w] + 1$
4) Set $\mathsf{addr} = F(K_T, w||\mathsf{UpdateCnt}[w]||0)$
5) Set $\mathsf{val} = (\mathsf{id}||\mathsf{op}) \oplus F(K_T, w||\mathsf{UpdateCnt}[w]||1)$
6) Set $\alpha = F_p(K_Y, \mathsf{id}||\mathsf{op}) \cdot (F_p(K_Z, w||\mathsf{UpdateCnt}[w]))^{-1}$
7) Set $\mathsf{xtag} = g^{F_p(K_X, w) \cdot F_p(K_Y, \mathsf{id}||\mathsf{op})}$
8) Send $(\mathsf{addr}, \mathsf{val}, \alpha, \mathsf{xtag})$ to the server

Server

1) Parse **EDB** $= (\mathsf{TSet}, \mathsf{XSet})$
2) Set $\mathsf{TSet}[\mathsf{addr}] = (\mathsf{val}, \alpha)$
3) Set $\mathsf{XSet}[\mathsf{xtag}] = 1$

Figure 8: ODXT. UPDATE $(\mathsf{sk}, \mathsf{st}, \mathsf{op}, (\mathsf{id}, w);$ **EDB**$)$

scheme of Lai *et al.* [9], a static cross tag was conceptually divided into two parts, one corresponding to the keyword w_i and the other corresponding to *only* the document identifier id_j. In ODXT, we additionally involve the operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ in the generation of the cross-tag, and combine it with the document identifier id_j. Similar to BDXT, this allows a cross-tag to indicate whether a certain operation (either addition or deletion) involving a given keyword-file pair has occurred or not, which in turn allows the XSet to be dynamic and forward privacy-preserving.

However, where we improve over BDXT is in achieving a stronger notion of backward privacy by minimizing leakages during searches, as discussed subsequently. A crucial role in this regard is played by the dynamic blinding factor α in ODXT, which can also be computed on-the-fly with every update operation. In other words, unlike OXT [9] and HXT'[29], we completely avoid the need for any pre-computation at setup. By involving the operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ in the generation of both the cross tags and the blinding factors, we now allow both the TSet and XSet to be updated dynamically in tandem while preserving forward privacy. In particular, our TSet now differs significantly from that in $\text{MITRA}_{\text{CONJ}}$ in its contents and also the manner in which it is updated. The concept of dynamic blinding factors does not appear in MITRA, or for that matter, any existing dynamic conjunctive SSE scheme.

As demonstrated subsequently, dynamic blinding factors additionally allow oblivious reconstruction of cross tags during conjunctive searches, which suppresses leakages and paves the way for strong backward privacy guarantees. Hence, the introduction of dynamic blinding factors is another novel technical contribution of this work.

Oblivious Conjunctive Search. We now elucidate the overall idea for oblivious conjunctive search. Unlike in BDXT, where the s-term in a conjunctive query was chosen to be the keyword with the least frequency, in ODXT, we choose the s-term to be the keyword *involved in the least number of update operations*. We note, however, that in real-life databases a keyword that occurs across fewer documents is also likely to be involved in fewer update operations, especially in systems where an update operation takes an entire file for addition/deletion. Additionally, the client no longer needs two separate data structures UpdateCnt and DBCnt to keep track of both the number of update operations involving a keyword and the

number of documents actually containing it.

Suppose that in a conjunctive query $q = (w_1 \wedge \ldots \wedge w_n)$, w_1 is the keyword *involved in the least number of update operations*. Let $(\mathsf{op}, (\mathsf{id}_j, w_1))$ be the cnt^{th} update operation involving w_1 and *suppose* that the server is able to compute each cross-tag $\mathsf{xtag}_{i,j,\mathsf{op}}$ for $w_i \in \{w_2, \ldots, w_n\}$. In that case, the server is able to check each such cross-tag for membership in the XSet dictionary, and let the client know the corresponding outcomes.

For example, if the cnt^{th} update operation was an *insert* operation, the client learns exactly how many insertion operations involving id_j and keywords among $w_1, \ldots, w_n$ have been executed so far. Similarly, if this was a *deletion* operation, the client learns exactly how many deletion operations involving id_j and keywords among $w_1, \ldots, w_n$ have been executed.

Once the client gets this information from the server, it can compute the final list of document identifiers as follows: among all document identifiers that appear in operations involving w_1, retain those that satisfy *both* of the following:

- It has been *inserted* for *every* keyword $w_1, \ldots, w_n$,
- It has *not* been *deleted* for *any* keyword $w_1, \ldots, w_n$

The challenge is to allow the server to compute the cross-tags obliviously, i.e., without explicitly learning the actual identifier-operation pair $(\mathsf{id}_j, \mathsf{op})$*, via a single message received from the client.*

Oblivious Cross-Tag Computation. To enable this, the client does the following: for the cnt^{th} update operation involving the keyword w_1, it sends to the server the corresponding TSet address (same as in BDXT) along with an additional (permuted) set of *cross-tokens* $\{\mathsf{xtoken}_{i,\mathsf{cnt}}\}_{i \in [n]}$ where for each $i \in [n]$, we have

$$\mathsf{xtoken}_{i,\mathsf{cnt}} = g^{F_p(K_X, w_i) \cdot F_p(K_Z, w_1 || \mathsf{cnt})}.$$

Now recall that the TSet address corresponding to the cnt^{th} update operation involving w_1 stores an additional precomputed blinding factor α, where

$$\alpha = F_p(K_Y, \mathsf{id}_j || \mathsf{op}) \cdot (F_p(K_Z, w_1 || \mathsf{cnt}))^{-1}.$$

It is easy to see that given a cross-token $\mathsf{xtoken}_{i,\mathsf{cnt}}$ and the blinding factor α, the server can compute the cross-tag as:

$$\mathsf{xtag}_{i,j,\mathsf{op}} = g^{F_p(K_X, w_i) \cdot F_p(K_Y, \mathsf{id}_j || \mathsf{op})} = (\mathsf{xtoken}_{i,\mathsf{cnt}})^{\alpha}.$$

In other words, without ever learning what the underlying identifier id_j or the underlying operation op was, the server *obliviously* computes the relevant cross-tag involving the keyword w_i and the pair $(\mathsf{id}_j, \mathsf{op})$. Note that we explicitly use the fact that $\mathsf{xtag}_{i,j,\mathsf{op}}$ and α share the same sub-terms pertaining to the pair $(\mathsf{id}_j, \mathsf{op})$ to enable this oblivious computation.

To see why this is useful, recall that in BDXT, the second round of communication between the client and the server essentially involved the client explicitly computing and sending across the relevant xtag values to the server. In ODXT, we save this additional round of communication by allowing the client and the server to engage in a specially designed single-round protocol where the server directly gets the xtag values.

Client

1) Parse $\mathsf{sk} = (K_T, K_X)$ and $\mathsf{st} = \mathsf{UpdateCnt}$
2) Use UpdateCnt to identify keyword with least updates (assumed to be w_1 w.l.o.g)
3) Initialize stokenList to an empty list
4) Initialize $\mathsf{xtokenList}_1, \ldots, \mathsf{xtokenList}_{\mathsf{UpdateCnt}[w_1]}$ to empty lists
5) **For** $j = 1$ **to** $\mathsf{UpdateCnt}[w_1]$:
   a) Set $\mathsf{saddr}_j = F(K_T, w_1 || j || 0)$
   b) Set $\mathsf{stokenList} = \mathsf{stokenList} \cup \{\mathsf{saddr}_j\}$
   c) **For** $i = 2$ **to** n:
      i) Set $\mathsf{xtoken}_{i,j} = g^{F_p(K_X, w_i) \cdot F_p(K_Z, w_1 || j)}$
      ii) Set $\mathsf{xtokenList}_j = \mathsf{xtokenList}_j \cup \{\mathsf{xtoken}_{i,j}\}$
   d) **End For**
   e) Randomly permute the tuple-entries of $\mathsf{xTagList}_j$
6) **End For**
7) Send $(\mathsf{stokenList}, \mathsf{xtokenList}_1, \ldots, \mathsf{xtokenList}_{\mathsf{UpdateCnt}[w_1]})$ to the server

Server

1) Parse $\mathbf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$
2) Initialize sEOpList to an empty list
3) **For** $j = 1$ **to** stokenList.size:
   a) Set $\mathsf{cnt}_j = 1$
   b) Set $(\mathsf{sval}_j, \alpha_j) = \mathsf{TSet}[\mathsf{stokenList}[j]]$
   c) **For** $i = 2$ **to** n:
      i) Set $\mathsf{xtoken}_{i,j} = \mathsf{xtokenList}_j[i]$
      ii) Compute $\mathsf{xtag}_{i,j} = (\mathsf{xtoken}_{i,j})^{\alpha_j}$
      iii) **If** $\mathsf{XSet}[\mathsf{xtag}_{i,j}] = 1$, **then** set $\mathsf{cnt}_j = \mathsf{cnt}_j + 1$
   d) **End For**
   e) Set $\mathsf{sEOpList} = \mathsf{sEOpList} \cup \{(j, \mathsf{sval}_j, \mathsf{cnt}_j)\}$
4) **End For**
5) Send sEOpList to the client

Client: Final Output Computation

1) Initialize IdList to an empty list
2) **For** $\ell = 1$ **to** sEOpList.$size$:
   a) Let $(j, \mathsf{sval}_j, \mathsf{cnt}_j) = \mathsf{sEOpList}[\ell]$
   b) Recover $(\mathsf{id}_j || \mathsf{op}_j) = \mathsf{sval}_j \oplus F(K_T, w_1 || j || 1)$
   c) **If** op_j is add **and** $\mathsf{cnt}_j = n$ **then** set $\mathsf{sIdList} = \mathsf{sIdList} \cup \{\mathsf{id}_j\}$
   d) **Else if** op_j is del **and** $\mathsf{cnt}_j > 0$ **then** set $\mathsf{sIdList} = \mathsf{sIdList} \setminus \{\mathsf{id}_j\}$
3) **End For**
4) Output IdList

Figure 9: ODXT. SEARCH $(\mathsf{sk}, \mathsf{st}, q = (w_1 \wedge \ldots \wedge w_n); \mathbf{EDB})$

The oblivious computation described above constitutes the core of this protocol. Beyond this, the rest of the search operation proceeds along the same lines as BDXT.

Putting these ideas together, we get the ODXT protocol, as described across Figures 7, 8 and 9.

Server Storage. The server stores the dictionaries TSet and XSet. Note that during setup, the TSet and XSet dictionaries are both initialized to be empty. After N updates, the storage requirement at the server grows linearly to $O(N\lambda)$, since each update operation adds a $O(\lambda)$-sized entry of the form $(\mathsf{addr}, \alpha, \mathsf{val})$ to TSet and a $O(\lambda)$-sized cross-tag entry of the form $(\mathsf{xtag}, 1)$ to XSet. In other words, the storage requirement at the server grows *linearly* with the number of update operations on the dataset. This is exactly as in the BDXT scheme described earlier.

Client Storage. ODXT approximately halves the local storage requirement at the client as compared to BDXT. In ODXT, the client locally stores only a single array UpdateCnt, as opposed to both UpdateCnt and DBCnt in BDXT. This makes

the client storage requirements for ODXT comparable to the naïve adaptation of MITRA, as well as other dynamic SSE schemes supporting single keyword search [5], [6], [10], [37].

Note that during setup, this array is initialized to empty. After N updates, the storage requirement at the client grows as $O(|\mathcal{W}| \cdot \log N)$, $|\mathcal{W}|$ is the size of the keyword dictionary, which is typically upper-bounded by some large pre-defined constant. In other words, the storage requirement at the client grows *logarithmically* with the number of update operations.

Search Performance. ODXT requires a *single* round of communication between the client and the server during conjunctive searches. The computational overheads at both the client and the server, as well as the communication overheads, scale with $O(n \cdot |\mathsf{Upd}(w_1)|)$. First of all, this is still a significant improvement over the naïve adaptation over MITRA whenever there is a query term in the conjunction with relatively small frequency of updates.

While searches in BDXT incur lower computational overhead in the asymptotic sense, it is worth observing that in real-life databases, a keyword that occurs across fewer documents is also likely to be involved in fewer update operations, especially in systems where an update operation takes an entire file for addition/deletion. So for real-life databases, the s-terms for BDXT and ODXT are likely to be the same for most conjunctive queries, and the number of updates on the s-term is unlikely to be significantly larger than the number of documents currently containing it.

D. Leakage Profile of ODXT (Informal)

We now present an informal overview of the leakage profile for ODXT.

Update Leakages. Updates in ODXT are leakage-free. This is because during updates, the server only sees a TSet (address, value) pair and a cross-tag, all of which are generated using PRFs and appear only once under the assumption that file identifiers are never repeated[4]. This in turn implies that ODXT is forward private.

Search Leakages. Next, we informally summarize the leakages incurred by ODXT during conjunctive searches.

Output Leakage: The server learns the final set of document identifiers in the conjunction, since we assume that the client sends these in the clear to retrieve the corresponding documents.

s-term Leakage: The server learns the number of update operations involving the s-term w_1, as well as the time stamp for each such operation.

Common s-Term Leakage: The server learns if two (or more) conjunctive queries have the same s-term w_1. This is because, for all queries where the s-term is w_1, the client sends across the same set of (or a superset of the same set of) stoken values corresponding to update records involving w_1 in the TSet dictionary.

[4]This assumption is made in several existing forward and backward private SSE schemes for single keyword search, most notably in the constructions of Bost *et al.* [6] and Chamani *et al.* [10], including the original MITRA scheme.

x-term Leakage: For each update operation $(\mathsf{op}_j, (\mathsf{id}_j, w_1))$ involving the s-term w_1, the server learns the total number of update operations of the form $(\mathsf{op}_j, (\mathsf{id}_j, w_i))$ for each x-term $w_i \in \{w_2, \ldots, w_n\}$, as well the corresponding time stamp for each such operation.

Common x-Term Leakage: The server learns if two queries with (possibly distinct) s-terms w_1 and w_1' share a common x-term w_i, provided that the update histories for w_1 and w_1' involve at least one common document identifier id_j. This is because when processing these queries, the server would encounter a common cross-tag $\mathsf{xtag}_{i,j}$.

Improvements over BDXT. It is easy to see that ODXT improves significantly over BDXT in terms of leakage. To begin with, in ODXT, the server does not learn the frequency of the s-term, i.e., $|\mathbf{DB}(w_1)|$; it only learns the number of update operations involving the s-term, i.e., $|\mathsf{Upd}(w_1)|$. This is exactly as in the naïve adaptation of MITRA to the conjunctive setting. On the other hand, in BDXT, the server learns both $|\mathsf{Upd}(w_1)|$ and $|\mathbf{DB}(w_1)|$.

Moreover, in ODXT, the server does not learn which cross-tag pairs in the XSet dictionary correspond to the *same* identifier-keyword pair. Learning this information would require the server to be able to correlate cross-tags generated across different update operations, which is computationally infeasible since the PRF F_p hides any such correlation. Consequently, it does not learn the exact number of keywords among $w_2, \ldots, w_n$ that each document in $\mathbf{DB}(w_1)$ contains. This is a major improvement over BDXT, where the server was able to learn this information.

E. Formalizing the Leakage Profile of ODXT

In this section, we formally describe the leakage profile for ODXT and prove its forward and backward privacy. Intuitively, a dynamic conjunctive SSE scheme is *forward and backward private* if: (a) an update operation reveals no additional information about a conjunctive search operation that took place at an earlier time, and (b) if a search operation on a conjunction $q = (w_1 \wedge \ldots \wedge w_n)$ reveals no information about certain deletion operations on $(w_1, \ldots, w_n)$ that took place at an earlier time. We formally establish below that ODXT achieves this notion of forward and backward privacy.

Let $\mathcal{Q}$ be a list with the following types of entries:

- (t, w): search query on keyword w at timestamp t.
- $(t, \mathsf{op}, (\mathsf{id}, w))$: update query $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ on identifier-keyword pair (id, w) at timestamp t.

Output Leakages. For any keyword w, we define $\mathsf{TimeDB}(w)$ to be the function that returns the list of all file identifiers containing w that have not yet been deleted, along with their respective timestamps of insertion. More formally, we have

$$\mathsf{TimeDB}(w) = \{(t, \mathsf{id}) \mid (t, \mathsf{add}, (\mathsf{id}, w)) \in \mathcal{Q} \quad \textit{and} \quad \forall t' : (t', \mathsf{del}, (\mathsf{id}, w)) \notin \mathcal{Q}\}$$

We overload notation to define $\mathsf{TimeDB}(q)$ for any conjunctive query $q = (w_1 \wedge \ldots \wedge w_n)$ as

$$\mathsf{TimeDB}(q) = \{(\{t_i\}_{i \in [n]}, \mathsf{id}) \mid (t_i, \mathsf{add}, (\mathsf{id}, w_i)) \in \mathcal{Q} \quad \textit{and} \quad \forall t' : (t', \mathsf{del}, (\mathsf{id}, w_i)) \notin \mathcal{Q}\}$$

In other words, $\mathsf{TimeDB}(q)$ returns the list of identifiers corresponding to documents containing $w_1, \ldots, w_n$ that have not yet been deleted, along with their respective timestamps of insertion. Intuitively, $\mathsf{TimeDB}(q)$ captures the output leakage for q.

s-Term Leakages. For any keyword w, we define $\mathsf{Upd}(w)$ to be the function that returns the timestamps of all update operations on w. More formally, we have

$$\mathsf{Upd}(w) = \{t \mid \exists(\mathsf{op}, \mathsf{id}) : (t, \mathsf{op}, (\mathsf{id}, w)) \in \mathcal{Q}\}.$$

Intuitively, for a conjunctive query $q = (w_1 \wedge \ldots \wedge w_n)$, where w_1 is the s-term, $\mathsf{Upd}(w_1)$ captures all s-term leakages for q.

x-Term Leakages. Next, we again overload notation to define $\mathsf{Upd}(w_1, w_2)$ for any pair of keywords (w_1, w_2) as

$$\mathsf{Upd}(w_1, w_2) = \{(t_1, t_2) \mid \exists(\mathsf{op}, \mathsf{id}) : (t_1, \mathsf{op}, (\mathsf{id}, w_1)) \in \mathcal{Q} \text{ and } (t_2, \mathsf{op}, (\mathsf{id}, w_2)) \in \mathcal{Q}\}$$

In other words, $\mathsf{Upd}(w_1, w_2)$ returns the timestamps of all update operations on w_1 and w_2 that involve the same document identifier. Intuitively, for a conjunctive query $q = (w_1 \wedge \ldots \wedge w_n)$, where w_1 is the s-term, $\{\mathsf{Upd}(w_1, w_i)\}_{i \in [n]}$ captures all x-term leakages for q.

For ease of representation, we combine the s-term and x-term leakages from a given query as follows: we further overload notation to define $\mathsf{Upd}(q)$ for $q = (w_1 \wedge \ldots \wedge w_n)$, where w_1 is the s-term, as

$$\mathsf{Upd}(q) = \mathsf{Upd}(w_1) \cup \left(\bigcup_{i=2}^{n} \mathsf{Upd}(w_1, w_i) \right).$$

ODXT **Leakage Profile.** We are now ready to formally define the leakage profile for ODXT as:

$$\mathcal{L}_{\text{ODXT}} = \left(\mathcal{L}^{\text{Setup}}_{\text{ODXT}}, \mathcal{L}^{\text{Search}}_{\text{ODXT}}, \mathcal{L}^{\text{Upd}}_{\text{ODXT}} \right),$$

where

- $\mathcal{L}^{\text{Setup}}_{\text{ODXT}} = \perp$.
- $\mathcal{L}^{\text{Upd}}_{\text{ODXT}}(\mathsf{op}, (\mathsf{id}, w)) = \perp$.
- $\mathcal{L}^{\text{Search}}_{\text{ODXT}}(q) = (\mathsf{TimeDB}(q), \mathsf{Upd}(q))$.

Finally, we state the following theorem for the security of ODXT.

Theorem-1 (Security of ODXT**).** *Assuming that F and F_p are secure PRFs and the decisional Diffie-Hellman assumption holds over the group $\mathcal{G}$,* ODXT *is adaptively-secure with respect to a leakage function $\mathcal{L}_{\text{ODXT}}$.*

The detailed proof appears in the full version of the paper [31] due to lack of space.

F. Forward Privacy of ODXT

In this section, we formally describe the forward privacy guarantees of ODXT. According to the formal definition introduced by Bost et al. [6], a dynamic conjunctive SSE scheme that is adaptively secure with respect to a leakage profile

$$\mathcal{L} = \left(\mathcal{L}^{\text{Setup}}, \mathcal{L}^{\text{Search}}, \mathcal{L}^{\text{Upd}} \right),$$

is said to be adaptively forward private if there exists a stateless function $\mathcal{L}'$ such that for any arbitrary triplet $(\mathsf{op}, \mathsf{id}, w)$, we have

$$\mathcal{L}^{\text{Upd}}(\mathsf{op}, (\mathsf{id}, w)) = \mathcal{L}'(\mathsf{op}, \mathsf{id}).$$

Intuitively, this captures the fact that an update operation computationally hides the underlying keyword w, and hence it cannot be correlated with any previous search query involving w by a computationally bounded adversary.

We now examine whether ODXT is forward private as per this definition. Since $\mathcal{L}^{\text{Upd}}_{\text{ODXT}}(\mathsf{op}, (\mathsf{id}, w)) = \perp$, an update operation in ODXT hides not only the underlying keyword w, but also the identifier id and the operation op. In other words, the following is a natural corollary of Theorem-1:

Corollary-1 (Forward Privacy of ODXT**).** *Assuming that F and F_p are secure PRFs and the decisional Diffie-Hellman assumption holds over the group $\mathcal{G}$,* ODXT *is adaptively forward private.*

G. Backward Privacy of ODXT

Next, we formally describe the backward privacy guarantees of ODXT. According to the formal definition introduced by Bost et al. [6], a dynamic SSE scheme that supports *single keyword searches* and is adaptively secure with respect to some leakage function $\mathcal{L} = \left(\mathcal{L}^{\text{Setup}}, \mathcal{L}^{\text{Search}}, \mathcal{L}^{\text{Upd}} \right)$ is adaptively *Type-II backward private* if there exist stateless functions $\mathcal{L}''$ and $\mathcal{L}'''$ such that for any $(\mathsf{op}, \mathsf{id}, w)$, we have

$$\mathcal{L}^{\text{Upd}}(\mathsf{op}, (\mathsf{id}, w)) = \mathcal{L}''(\mathsf{op}, \mathsf{id})), \quad \text{and}$$
$$\mathcal{L}^{\text{Search}}(w) = \mathcal{L}'''(\mathsf{TimeDB}(w), \mathsf{Upd}(w)).$$

We now examine whether ODXT is forward backward as per this definition. Recall that we have

$$\mathcal{L}^{\text{Upd}}_{\text{ODXT}}(\mathsf{op}, (\mathsf{id}, w)) = \perp, \mathcal{L}^{\text{Search}}_{\text{ODXT}}(q) = (\mathsf{TimeDB}(q), \mathsf{Upd}(q)),$$

for any conjunctive query q. This is a natural generalization of the aforementioned leakage profile for Type-II backward privacy from the setting of single keyword searches to our setting of conjunctive keyword searches. Hence, the following is also a natural corollary of Theorem-1:

Corollary-2 (Backward Privacy of ODXT**).** *Assuming that F and F_p are secure PRFs and the decisional Diffie-Hellman assumption holds over the group $\mathcal{G}$,* ODXT *is adaptively Type-II backward private.*

H. Discussion on the Leakage Profile of ODXT

In this subsection, we present a more in-depth analysis of the leakage profile for ODXT during conjunctive searches and its implications.

Output Leakage. We begin by noting that the output leakage (alternatively, the result pattern leakage) is incurred by nearly all existing SSE schemes, including static and dynamic schemes, in the setting of both single and conjunctive keyword searches (such as in [14], [9], [29], [6], [10], [37]). This is usually considered acceptable in the SSE literature; indeed the few known data/query recovery attacks that manage to exploit this leakage ([22], [7], [38], [3]) assume extremely strong

adversarial models where the adversary has partial knowledge of the plaintext database/queries.

s-Term Leakages. We focus next on the leakages related to the s-term, namely, the total number of operations on the s-term and the timestamps corresponding to these operations. We begin by noting that these leakages are somewhat inherent in our design paradigm, which attempts to tie both the search complexity and the leakage to the s-term, as it has the least frequency of occurrence. We draw parallels with conjunctive SSE schemes in the static setting, most notably the scheme of Cash *et al.* [9] and the more recent scheme of Lai *et al.* [29], which incur similar s-term leakages.

In the setting of single keyword search, existing forward and Type-II backward private SSE schemes [6], [10], [37] also incur leakages of update patterns; the only constructions not to incur such leakages seem to rely on the use of ORAM-style data structures [6], [10]. Fortifying ODXT with such data structures in an attempt to prevent this leakage is an interesting open challenge, although this would probably have to trade-off with some degradation in search performance (mostly in terms of communication complexity and number of rounds of communication during searches).

It is also possible (and perhaps conceptually simpler) to mask this leakage by using volume-hiding techniques such as padding [14], [25] where for the s-term w_1, the client additionally sends a randomly chosen set of dummy stoken keys to the server, such that the total number of stoken keys sent is the same for all queries. This would incur a degradation in search performance, and it is up to the designer to decide on a suitable trade-off between performance and leakage.

However, we would like to point out that there are no known data/query recovery attacks on either static or dynamic conjunctive SSE schemes that specially exploit leakages related to the s-term. So we believe that *even without the aforementioned fortifications*, it appears that our ODXT scheme is not vulnerable to any known attacks due to the leakages related to the s-term.

x-Term Leakages. Next, we focus on the x-term leakages. We again draw parallels with conjunctive SSE schemes in the static setting, most notably the scheme of Cash *et al.* [9] and the more recent scheme of Lai *et al.* [29], which incur similar x-term leakages. The only known attack on conjunctive SSE schemes that exploits a form of x-term leakages is the *file injection attack* proposed by Zhang *et al.* in [38]. More concretely, the adversarial server must be able to compute $|\mathbf{DB}(w_1) \cap \mathbf{DB}(w_i)|$ when processing the search query.

We note however that for file injection attacks to work efficiently, the adversarial server must recover, for every x-term w_i, the result size corresponding to each sub-query of the form $w_1 \cap w_i$. However, the x-term leakage profile of ODXT is not sufficient to compute this term, since the set of xtoken values sent to the server is randomly permuted precisely to mask such inference-style attacks. In addition, in ODXT, the server only learns update histories, and not the exact correspondences between insertions and deletions on the same identifier-keyword pair, which is also necessary for inferring the aforementioned information.

Once again, either implementing the XSet using ORAM-style data structures or adopting volume-hiding techniques such as padding may be useful in masking this leakage even further; however, even without such additional fortifications, it appears that our ODXT scheme is not vulnerable to file injection attacks, or any other known attacks for that matter, due to the leakages related to the x-terms in a conjunctive query.

I. ODXT in the Multi-Client Setting

As already discussed, ODXT removes the need for an additional round of communication between the client and the server during conjunctive searches. Beyond the obvious savings in terms of search latency, this also potentially expands the applicability of ODXT to settings where multiple rounds of interaction are unsuitable, such as the multi-client SSE setting.

In the multi-client setting, a data owner outsources its encrypted data to an external server and enables other parties to perform queries on the encrypted data by providing them with search tokens for specific queries. The key requirement is that external parties should learn no information beyond what is revealed by the search tokens authorized to them.

Unfortunately, schemes such as BDXT with search operations involving multiple rounds of client-server communication are inherently unsuited to the multi-client setting. This is because such schemes potentially allow the untrusted server to collude with malicious clients and recover sensitive information about queries issued by honest clients [9]. In particular, a malicious client could gain access to intermediate messages exchanged between the server and the honest clients, and exploit them to learn outcomes of queries involving conjuncts that it was not originally authorized for.

On the other hand, ODXT involves a single round of communication during searches. Hence, it is inherently resistant to such attacks. In particular, since the only message from the server to each client is the final list of file identifiers corresponding to the client's query, there are no intermediate messages that a malicious client could observe/manipulate to infer unauthorized information. Consequently, ODXT can be combined with well-established authorization techniques for *controlled disclosure* (such as discussed in [12], [27], [23]) and deployed in the multi-client setting. Additionally, using techniques introduced by the authors of [23], ODXT can be extended to hide client-issued queries not only from the server but also from the token issuing authority.

As a concrete example, when ODXT is implemented in the multi-client setting, the token generation algorithm can be implemented using a secure two-party oblivious transfer (OT) protocol [1], [30] between the client and the token issuing authority. For simplicity, we can assume that the token issuing authority is the data owner itself (the same assumption is made in [23].

In this protocol, the data owner's input would be the secret key used to generate search tokens, while the client's input would be the keyword(s) that is wishes to search for. At the end of the protocol, the client would learn the search token(s) corresponding to its query without gaining any additional information about the secret key, while the data owner would

learn no information about the query issued by the client. After this, the client can simply forward this search token to the server, and the search process would be executed exactly as in the ODXT protocol described in Section III-C. We can also argue that this affords the client precisely the same query privacy guarantees against the server as the original ODXT protocol.

We would also implement an authentication mechanism that would allow the server to verify that any search token that it receives from a client was actually issued by the data owner, and was not forged by the client. This is important to prevent query privilege escalation attacks wherein a client could try and issue queries beyond those authorized by the data owner. Since we are in the semi-honest setting, any standard authentication mechanism (e.g., existentially unforgeable digital signatures) would suffice for this purpose.

Finally, using techniques from [23], we can also boost the security of ODXT in the multi-client setting to withstand arbitrarily malicious behavior from both the data owner as well as from a group of (potentially colluding) clients. Such techniques would not compromise the core security and efficiency guarantees of ODXT.

IV. Experimental Evaluation

In this section, we report on a prototype implementation of ODXT and compare it with a prototype implementation of $\text{MITRA}_{\text{CONJ}}$, which is a naïve adaptation of the MITRA scheme for conjunctive queries, as well as prototype implementations of dynamic variants of IEX-2LEV and IEX-ZMF proposed by Kamara and Moataz [24], which are *not* backward private.

Implementation Details. Our prototype implementations are developed in Python (version-3.8) using the PyCrypto library[5] for symmetric-key operations and the Sagemath library[6] for group-based operations. More specifically, we realize all PRF operations using AES-256 in counter mode, and all group operations in ODXT over the elliptic curve Curve25519 [2]. We implement the TSet data structure using Riak[7], which provides APIs for realizing distributed NoSQL key-value dictionaries, while the XSet dictionary is realized using a Bloom filter [4].

Platform and Dataset Used. For our experiments, we used a cluster of four 64-bit Intel Xeon E5-2690 v4 2.60GHz processors, running Ubuntu 18.04.1 LTS, with 128GB RAM and 1TB SSD hard disk, connected over a 10MBps wide-area network (WAN).

We used a 60.92GB-sized real world dataset from Wikimedia downloads [17], with 16 million documents and 43 million keywords. We simulated updates by randomly inserting and deleting documents from the original dataset into an empty dataset. Overall, we performed a total of 10^8 update operations, 30% of which were deletions. Our experiments were designed to ensure that each file in the 61GB dataset was inserted at least once; hence the entire database was effectively used.

[5]https://pycryptodome.readthedocs.io/en/latest/

[6]http://www.sagemath.org/

[7]http://basho.com/products/riak-kv/

A. Performance Evaluation

Multi-Threaded Implementations. Our experiments use multi-threaded implementations of the client and the server. In particular, for $\text{MITRA}_{\text{CONJ}}$, the search operation corresponding to each keyword in the queried conjunction is executed in parallel. Hence, the *search latency* for $\text{MITRA}_{\text{CONJ}}$ in our experiments is determined purely by the frequency of the *most frequent* keyword(s). Similarly, for ODXT, the search operations corresponding to the x-terms are executed in parallel; however by design, the *search latency* in our experiments depends only on the frequency of the *least frequent* keyword.

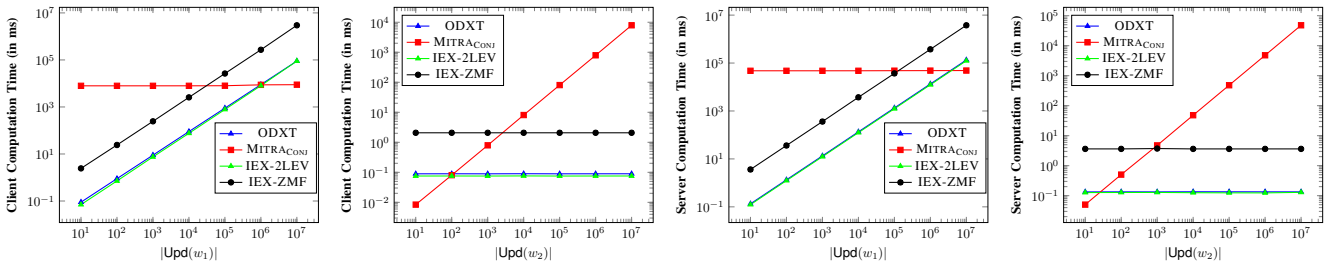
Search Latency v/s Computational Complexity. Note that in the setting of multi-threaded implementations, the variation of search latency with the frequency of keywords in the queried conjunction do not exactly correspond to the asymptotic expressions for computational overhead mentioned in Sections III-A and III-C. In particular, the expressions for computational overhead take into account the *total work done* across all the keywords in the conjunction. Nonetheless, the core advantage of ODXT over $\text{MITRA}_{\text{CONJ}}$ is also reflected in our experiments evaluating search latency.

Client and Server Latency. Figures 10 and 11 compare the various schemes with respect to the computational overheads at the client and the server for conjunctive searches involving two and six keywords, respectively. ODXT closely matches IEX-2LEV (despite achieving stronger security guarantees) and outperforms $\text{MITRA}_{\text{CONJ}}$ and IEX-ZMF in most cases. The only cases where $\text{MITRA}_{\text{CONJ}}$ either matches or outperforms ODXT is when all terms in the conjunction have nearly the same frequency, i.e., either the s-term has very high frequency of updates, or all x-terms have very low frequency of updates. However, such queries occur relatively rarely in practice. For most commonly encountered queries, ODXT offers significantly faster searches.

A simple observation is that in the extreme cases, the performance for ODXT can be boosted by using only the TSet to search for every keyword in the conjunction in parallel. This eliminates the usage of the heavier elliptic machinery, and achieve performance comparable with $\text{MITRA}_{\text{CONJ}}$. We illustrate this when we compare the end-to-end search latency of ODXT with the other benchmarks in Figure 13.

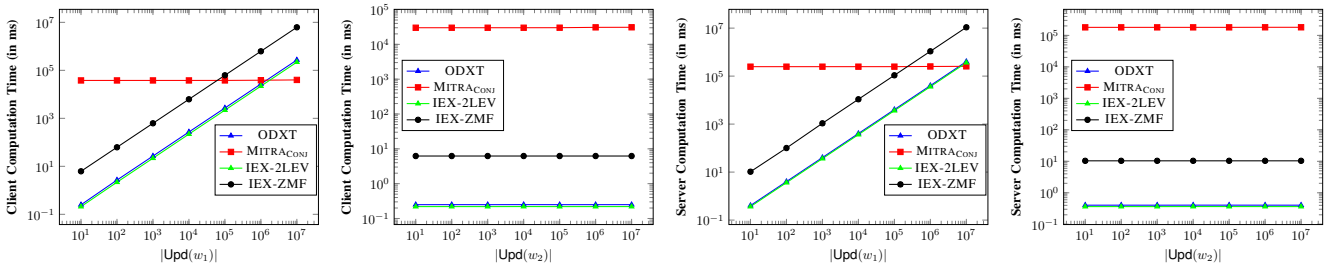
Communication Overheads. Figure 12 compares the various schemes with respect to the communication overheads for conjunctive searches involving two and multiple keywords, respectively. For ODXT and IEX-ZMF, the communication overheads scale with the update-frequency for the least frequent keyword, while in $\text{MITRA}_{\text{CONJ}}$, the communication overheads grow *cumulatively* with the frequency of *each* queried keyword. Note that IEX-ZMF has a constant communication overhead, but as we show later in Figure 14, this is achieved at the cost of nearly $100x$ greater storage as compared to ODXT.

Note: Observe that the flat lines corresponding to $\text{MITRA}_{\text{CONJ}}$ in Figure 12 have some "bumps" when the frequency of w_1 jumps from 10^6 to 10^7. For the two-keyword case, this is explained as follows: since the queries for w_1 and w_2 are executed in parallel, the contributions of w_1 and w_2 towards the overall communication overhead are proportional to their



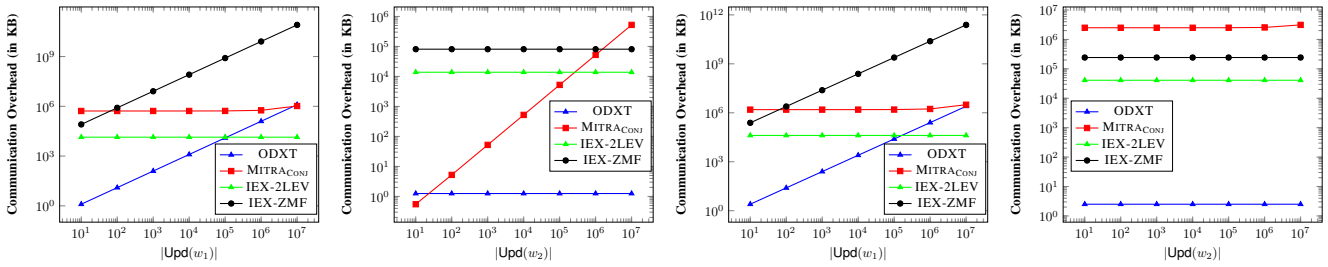
(a) $|\mathsf{Upd}(w_2)| = 10^7$ (b) $|\mathsf{Upd}(w_1)| = 10$ (c) $|\mathsf{Upd}(w_2)| = 10^7$ (d) $|\mathsf{Upd}(w_1)| = 10$

Figure 10: Two-conjunctive search query $q = (w_1 \wedge w_2)$: (a) computation time v/s variable $|\mathsf{Upd}(w_1)|$ (Client), (b) computation time v/s variable $|\mathsf{Upd}(w_2)|$ (Client), (c) computation time v/s variable $|\mathsf{Upd}(w_1)|$ (Server), and (d) computation time v/s variable $|\mathsf{Upd}(w_2)|$ (Server). The only cases where MITRA$_{\text{CONJ}}$ either matches or outperforms ODXT is when all terms in the conjunction have nearly the same frequency, i.e., either the s-term has very high frequency of updates, or all x-terms have very low frequency of updates. However, such queries relatively rarely in practice. For most commonly encountered queries, ODXT offers significantly faster searches.



(a) $|\mathsf{Upd}(w_2)| = 10^7$ (b) $|\mathsf{Upd}(w_1)| = 10$ (c) $|\mathsf{Upd}(w_2)| = 10^7$ (d) $|\mathsf{Upd}(w_1)| = 10$

Figure 11: Multi-conjunctive search query $q = (w_1 \wedge \ldots \wedge w_6)$ with $|\mathsf{Upd}(w_\ell)| = 10^7$ for $\ell \in [3, 6]$: (a) computation time v/s variable $|\mathsf{Upd}(w_1)|$ (Client), (b) computation time v/s variable $|\mathsf{Upd}(w_2)|$ (Client), (c) computation time v/s variable $|\mathsf{Upd}(w_1)|$ (Server), and (d) computation time v/s variable $|\mathsf{Upd}(w_2)|$ (Server)



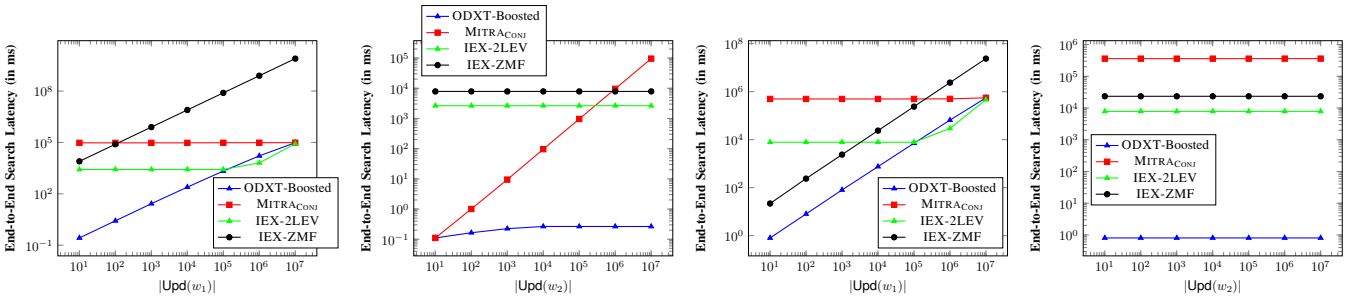
(a) $|\mathsf{Upd}(w_2)| = 10^7$ (b) $|\mathsf{Upd}(w_1)| = 10$ (c) $|\mathsf{Upd}(w_2)| = 10^7$ (d) $|\mathsf{Upd}(w_1)| = 10$

Figure 12: Two-conjunctive search query $q = (w_1 \wedge w_2)$: (a) communication overhead v/s variable $|\mathsf{Upd}(w_1)|$, (b) communication overhead v/s variable $|\mathsf{Upd}(w_2)|$. Multi-conjunctive search query $q = (w_1 \wedge \ldots \wedge w_6)$ with $|\mathsf{Upd}(w_\ell)| = 10^7$ for $\ell \in [3, 6]$: (c) communication overhead v/s variable $|\mathsf{Upd}(w_1)|$, (d) communication overhead v/s variable $|\mathsf{Upd}(w_2)|$

respective update-frequencies. The bumps indicate a transition point between "small" and "large" update-frequencies of w_1, relative to the update-frequency of w_2.

End-to-End Search Latency. Figure 13 compares ODXT (with the modification as mentioned above for boosting search performance in extreme cases) and the other schemes with respect to their end-to-end latency for conjunctive searches involving two and multiple keywords over a 10MBps wide-area network (WAN). As in our micro-benchmarks involving *only* the client or the server, the end-to-end search latency for ODXT scales only with the update-frequency of the least frequent keyword. It also outperforms all of the remaining schemes across queries involving keywords from all frequency ranges. In particular, the modification proposed above allows ODXT to be competitive with MITRA$_{\text{CONJ}}$ even in the extreme cases where all terms in the conjunction have nearly the same frequency.

Note that the end-to-end conjunctive search latency for ODXT is *less than* 10 *seconds* even when the frequency of the least frequent keyword w_1 is as high as 10^5. For example, the average end-to-end search latency of a conjunctive query of the form *"Find all emails containing the keywords stock, consensus, infrastructure and cash"* over a database of size 60.92GB is only 0.75 seconds, which is comparable in practice with the search latency over plaintext databases.



(a) $|\mathsf{Upd}(w_2)| = 10^7$ (b) $|\mathsf{Upd}(w_1)| = 10$ (c) $|\mathsf{Upd}(w_2)| = 10^7$ (d) $|\mathsf{Upd}(w_1)| = 10$

Figure 13: Experimental results with boosted ODXT: Two-conjunctive search query $q = (w_1 \wedge w_2)$: (a) end-to-end search latency v/s variable $|\mathsf{Upd}(w_1)|$, (b) end-to-end search latency v/s variable $|\mathsf{Upd}(w_2)|$. Multi-conjunctive search query $q = (w_1 \wedge \ldots \wedge w_6)$ with $|\mathsf{Upd}(w_\ell)| = 10^7$ for $\ell \in [3, 6]$: (c) end-to-end search latency v/s variable $|\mathsf{Upd}(w_1)|$, (d) end-to-end search latency v/s variable $|\mathsf{Upd}(w_2)|$

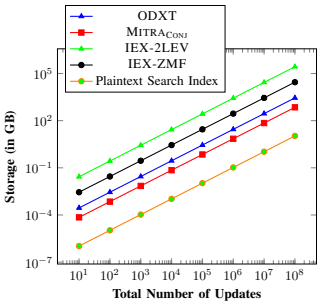


Figure 14: Server storage v/s number of updates

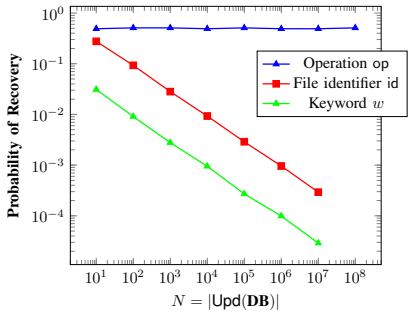


Figure 15: Leakage Analysis of ODXT: Updates in the "Known Update" Setting

Storage Overheads. Figure 14 compares the schemes with respect to the storage overhead at the server. In all cases, the storage overhead grows linearly with the number of updates. The storage overhead for MITRA$_{CONJ}$ is approximately $75x$ that of the plaintext search index. Despite the additional storage required for the XSet dictionary, ODXT requires only $3x$ more storage compared to MITRA$_{CONJ}$, which seems to be a reasonable trade-off for the vast improvements in search performance. Finally, the storage overheads for IEX-ZMF and IEX-2LEV are $10x$ and $100x$ larger than that for ODXT.

B. Leakage Analysis

In this section, we experimentally analyze the leakage profile of ODXT. The experiments were conducted over the same 60.92GB-sized real world dataset from Wikimedia downloads [17] as was used for the performance evaluation experiments in Section IV. Recall that the dataset contains 16 million documents and 43 million keywords.

Leakage Evaluation of Updates. We first present leakage evaluation of the update protocol in ODXT. We evaluate the probability that the adversary guesses correctly either the operation op or the document identifier id or the keyword w underlying a given update operation. As stated earlier in footnote 3, our leakage enumeration works under the assumption that update operations involving the same file identifier are never repeated. In particular, when an existing file is to be updated, it is deleted and re-inserted (in modified form) under a fresh file identifier. This assumption is made in several existing forward and backward private SSE schemes for single keyword search, most notably in the constructions of Bost *et al.* [6] and Chamani *et al.* [10], including the original MITRA scheme.

We design our experimental evaluation of the leakages from updates based on the assumption.

Our first set of experiments are in the "known update" model. More specifically, we assume that any given point of time, a computationally bounded adversary has seen $N = |\mathsf{Upd}(\mathbf{DB})|$ update tokens - each corresponding to an update operation involving unique $(\mathsf{op}, \mathsf{id})$ pair, and is trying to guess the operation op, the document identifier id and the keyword w underlying the next update operation. However, the adversary is not allowed to choose the update operations for which the tokens are to be generated. The adversary's knowledge is allowed to grow in a cumulative manner in that for each new update operation, the adversary is allowed to learn the underlying $(\mathsf{op}, (\mathsf{id}, w))$ tuple *after* fails to correctly predict the same.

Figure 15 illustrates the success probability of the adversary in this experiment as the number of tokens it has seen grows from 1 to 10^8. The results establish that the adversary can do no better than a "random guess". In particular, the guessing probability of the operation remains very close to 0.5 throughout (indicating an equal probability of addition and deletion), while the guessing probability for the file identifiers and keywords go down as the total number of files and keywords in the database grow with each update operation.

Our second set of experiments are in the "chosen update" model. More specifically, we assume that any given point of time, a computationally bounded adversary has seen $N = |\mathsf{Upd}(\mathbf{DB})|$ update tokens - each corresponding to an update

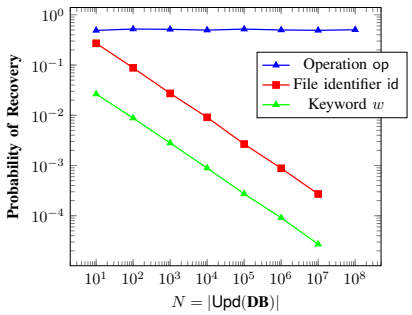


Figure 16: Leakage Analysis of ODXT: Updates in the "Chosen Update" Setting

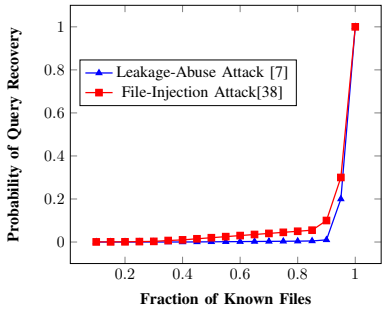


Figure 17: Leakage Analysis of ODXT: Two-Keyword Conjunctive Searches in the "Known Files" Setting

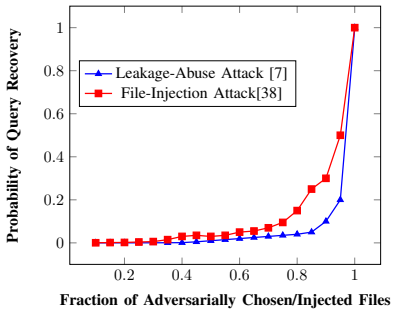


Figure 18: Leakage Analysis of ODXT: Two-Keyword Conjunctive Searches in the "Chosen Files" Setting

operation involving unique but *adversarially chosen* $(\mathsf{op}, \mathsf{id})$ pair and is trying to guess the operation op, the document identifier id and the keyword w underlying a fresh randomly chosen update operation.

Figure 16 illustrates the success probability of the adversary in this experiment as the number of chosen updates from 1 to 10^8. The results establish that even in this stronger setting, the adversary can do no better than a "random guess". In particular, the guessing probability of the operation again remains very close to 0.5 throughout (indicating an equal probability of addition and deletion), while the guessing probability for the file identifiers and keywords again go down as the total number of files and keywords in the database grow with each update operation.

Our experiments thus re-establish our formal statement that updates in ODXT computationally hide the underlying operation op, the file identifier id and the keyword w from the honest-but-curious adversarial server.

Leakage Evaluation of Conjunctive Searches. We now evaluate the leakage from the conjunctive search protocol in ODXT. We evaluate the probability that the adversary guesses correctly the keywords w_1 and w_2 underlying a two-conjunction query $q = (w_1 \wedge w_2)$ by one of two well-known and extensively studied cryptanalysis methodologies in the SSE literature- the *leakage-abuse attack* of Cash *et al.* [7] and the *file-injection attack* of Zhang *et al.* [38]. These attacks operate in two models - the *known* file model (where the adversary knows the contents of a certain fraction of the files in the database) and the *chosen/injected* file model (where a certain fraction of the files in the database are adversarially generated).

Naturally, when the adversary knows (or has injected) all the documents in the database, query recovery is trivial. However, this is a very strong attack model and is practicaly infeasible. What we want in a real-life application is that when the adversary knows only a small fraction of the files in the database, or has managed to inject a small fraction of files into the database, query recovery should happen with a very small probability. This would essentially indicate that the adversary has access to no additional leakage (about either the keywords underlying the query or the files in the database) from the search protocol beyond the benign leakage profile that was formally enumerated in Appendix III-E.

Figure 17 illustrates the success probability of the adversary for both kinds of attacks in the "known file" attack setting. The results clearly establish that even when the fraction of known files in the database is as high as 50%, the success probability of the adversary in recovering the keywords underlying a conjunction $(w_1 \wedge w_2)$ is less than 5%.

Similarly, figure 17 illustrates the success probability of the adversary for both kinds of attacks in the "chosen/injected file" attack setting. The results again establish that even when the fraction of injected files in the database is as high as 60% (which is quite unlikely in any real world database), the success probability of the adversary in recovering the keywords underlying a conjunction $(w_1 \wedge w_2)$ is less than 5%.

Our experiments thus re-establish our claims in Section III-E that the leakages incurred by the conjunctive search protocol in ODXT are benign and are resistant to even the most powerful leakage-based cryptanalysis techniques in the SSE literature over real-world databases.

V. Conclusion and Open Problems

In this work, we proposed the first dynamic SSE scheme supporting conjunctive keyword search that achieves *both* forward and backward privacy. Prior to this work, the study of forward and backward private SSE was restricted almost exclusively to single keyword search. On the other hand, in the setting of conjunctive keyword search, most prior SSE constructions with sub-linear search complexity only supported static databases.

Our main construction, called Oblivious Cross-Tags (ODXT in short), supports both updates and conjunctive keyword searches in tandem over very large arbitrarily-structured databases, including both attribute-value and free-text databases. All operations in ODXT involve only a single round of communication between the client and the server. This makes it amenable to deployment in a variety of settings such as single-client and multi-client SSE. Updates in ODXT are leakage-free, while searches incur only moderate access pattern leakages to the server that conform to existing notions of forward and backward privacy.

Our work gives rise to a number of interesting open problems. We leave it open to design dynamic conjunctive SSE schemes with even smaller leakage profiles. For example, an attractive goal is to construct a scheme that only reveals the update history pertaining to the final query outcome, and hides all the information pertaining to the least frequent keyword. Extending ODXT beyond conjunctions to support general Boolean queries is an interesting direction of future work. Finally, we leave open the question of achieving forward and backward private SSE schemes with (quasi-) optimal conjunctive keyword search complexity (along the lines of ORION and HORUS in [10]).

ACKNOWLEDGMENT

The second author would like to thank the grant "Design and Implementation of Efficient and Secure Searchable Encryption" sponsored by MHRD-STARS (Scheme for Transformational and Advanced Research in Sciences), India for partially supporting the work.

REFERENCES

[1] W. Aiello, Y. Ishai, and O. Reingold, "Priced oblivious transfer: How to sell digital goods," in *EUROCRYPT 2001*, 2001, pp. 119–135.

[2] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *PKC 2006*, 2006, pp. 207–228.

[3] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks," in *NDSS 2020*, 2020.

[4] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[5] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *ACM CCS 2016*, 2016, pp. 1143–1154.

[6] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *ACM CCS 2017*, 2017, pp. 1465–1482.

[7] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *ACM CCS 2015*, 2015, pp. 668–679.

[8] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *NDSS 2014*, 2014.

[9] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO 2013*, 2013, pp. 353–373.

[10] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *ACM CCS 2018*, 2018, pp. 1038–1055.

[11] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *ACNS 2005*, 2005, pp. 442–455.

[12] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *ASIACRYPT 2010*, 2010, pp. 577–594.

[13] C. Chu, W. T. Zhu, J. Han, J. K. Liu, J. Xu, and J. Zhou, "Security concerns in popular cloud storage services," *IEEE Pervasive Computing*, vol. 12, no. 4, pp. 50–57, 2013.

[14] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *ACM CCS 2006*, 2006, pp. 79–88.

[15] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans, "Efficient dynamic searchable encryption with forward privacy," *PoPETs*, vol. 2018, no. 1, pp. 5–20, 2018.

[16] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *ESORICS 2015*, 2015, pp. 123–145.

[17] W. Foundation, "Wikimedia downloads," https://dumps.wikimedia.org, 2017.

[18] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption," in *CRYPTO 2016*, 2016, pp. 563–592.

[19] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM STOC'09*, 2009, pp. 169–178.

[20] E. Goh, "Secure indexes," *IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003.

[21] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[22] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *NDSS 2012*, 2012.

[23] S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Outsourced symmetric private information retrieval," in *ACM CCS 2013*, 2013, pp. 875–888.

[24] S. Kamara and T. Moataz, "Boolean searchable symmetric encryption with worst-case sub-linear complexity," in *EUROCRYPT 2017*, 2017, pp. 94–124.

[25] ——, "Computationally volume-hiding structured encryption," in *EUROCRYPT 2019*, 2019, pp. 183–213.

[26] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *FC 2013*, 2013, pp. 258–274.

[27] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *ACM CCS 2012*, 2012, pp. 965–976.

[28] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *ACM CCS 2017*, 2017, pp. 1449–1463.

[29] S. Lai, S. Patranabis, A. Sakzad, J. K. Liu, D. Mukhopadhyay, R. Steinfeld, S. Sun, D. Liu, and C. Zuo, "Result pattern hiding searchable encryption for conjunctive queries," in *ACM CCS 2018*, 2018, pp. 745–762.

[30] M. Naor and B. Pinkas, "Efficient oblivious transfer protocols," in *SODA 2001*, 2001, pp. 448–457.

[31] S. Patranabis and D. Mukhopadhyay, "Forward and backward private conjunctive searchable symmetric encryption," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 1342, 2020.

[32] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *ACM SOSP 2011*, 2011, pp. 85–100.

[33] D. X. Song, D. A. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *IEEE S&P 2000*, 2000, pp. 44–55.

[34] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward private searchable symmetric encryption with optimized I/O efficiency," *IACR Cryptology ePrint Archive*, vol. 2018, p. 497, 2018.

[35] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *NDSS 2014*, 2014.

[36] S. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen, "An efficient non-interactive multi-client searchable encryption with support for boolean queries," in *ESORICS 2016*, 2016, pp. 154–172.

[37] S. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *ACM CCS 2018*, 2018, pp. 763–780.

[38] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *USENIX Security Symposium 2016*, 2016, pp. 707–720.