Searchable Encryption for Conjunctive Queries with Extended Forward and Backward Privacy

Cong Zuo*
Beijing Institute of Technology
Beijing, China
zuocong10@gmail.com

Shangqi Lai†
Monash University/CSIRO Data61
Clayton, Victoria, Australia
shangqi.lai@csiro.au

Shi-Feng Sun
Shanghai Jiao Tong University
Shanghai, China
shifeng.sun@sjtu.edu.cn

Xingliang Yuan
The University of Melbourne
Melbourne, Victoria, Australia
xingliang.yuan@unimelb.edu.au

Joseph K. Liu
Monash University
Clayton, Victoria, Australia
joseph.liu@monash.edu

Jun Shao
Zhejiang Gongshang University
Hangzhou, China
chn.junshao@gmail.com

Huaxiong Wang
Nanyang Technological University
Singapore
hxwang@ntu.edu.sg

Liehuang Zhu
Beijing Institute of Technology
Beijing, China
liehuangz@bit.edu.cn

Shujie Cui
Monash University
Clayton, Victoria, Australia
shujie.cui@monash.edu

Abstract

Recent developments in the field of Dynamic Searchable Symmetric Encryption (DSSE) with forward and backward privacy have attracted much attention from both research and industrial communities. However, most DSSE schemes with forward and backward privacy schemes only support single keyword queries, which impedes its prevalence in practice. Although some forward and backward private DSSE schemes with expressive queries (e.g., conjunctive queries) have been introduced, their backward privacy either essentially corresponds to single keyword queries or forward privacy is not comprehensive. In addition, the deletion of many DSSE schemes is achieved by addition paired with a deletion mark (i.e., lazy deletion). To address these problems, we present two novel DSSE schemes with conjunctive queries (termed SDSSE-CQ and SDSSE-CQ-S), which achieve both forward and backward privacy. To analyze their security, we present two new levels of backward privacy (named Type-O and Type-O$^-$, more and more secure), which give a more comprehensive understanding of the leakages of conjunctive queries in the OXT framework. Eventually, the security analysis and experimental evaluations show that the proposed schemes achieve better security with reasonable computation and communication increase.

Keywords

Forward Privacy, Backward Privacy, Dynamic Searchable Symmetric Encryption, Conjunctive Queries

*This research was partially done while Cong Zuo was affiliated with Nanyang Technological University.
†This research was done while Shangqi Lai was affiliated with Monash University.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–16

https://doi.org/XXXXXXX.XXXXXXX

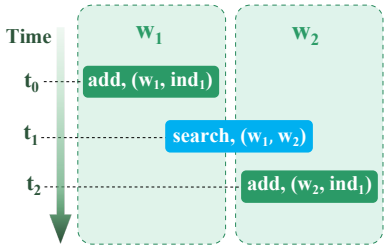
1 Introduction

Dynamic searchable symmetric encryption (DSSE) enables the update of the encrypted database while maintaining searchability, which is a useful tool for protecting users' data stored on the cloud. However, the update operations would cause more leaked information, which can be abused by the attackers [2, 7, 59]. In particular, Zhang et al. [59] proposed file-injection attacks, which can recover users' queries through update. Specifically, the attacker (i.e., the server) can inject some well-manipulated files. Then the search keywords can be recovered by checking if the injected files have been returned. To mitigate the attacks, DSSE schemes are required to hold two new security notions, namely forward privacy and backward privacy, which are introduced by Stefanov et al. [51]. Informally, forward privacy requires that the server cannot match newly updated files to previously issued search queries. Correspondingly, backward privacy does not allow the server to learn the files that were previously added and later deleted [1]. The formal definition of forward privacy and backward privacy for single keyword queries are proposed by Bost [5] and Bost et al. [6], respectively. Due to its complexity, Bost et al. [6] gave three levels of backward privacy, namely Type-III to Type-I, from the weakest to the strongest. Since then, many DSSE schemes with forward and/or backward privacy have been introduced [1, 12, 19, 38, 54, 55, 62].

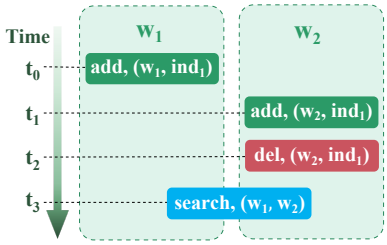
Nevertheless, most existing forward and backward private DSSE schemes only support single keyword queries, which is not always applicable in practice. As stated in [9, 34, 49], more expressive queries such as conjunctive queries are often desired. It is not an easy job to design forward and backward private DSSE supporting conjunctive queries, as we need to consider not only the leakages of each keyword in a conjunctive query but also the leakages of the conjunction of the keywords in the conjunctive query. The forward and backward privacy of DSSE for conjunctive queries

[1]As far as we know, currently, there are no specific attacks that can utilize the leakages described in backward privacy, while they could be used by attackers in the future. Besides, as many prior works, such as the original OXT paper [9], pointed out, one major goal of SSE designs is to minimize the leakages with reasonable efficiency trade-offs.

is essentially single keyword queries [28]. Recently, Patranabis et al. [49] proposed a DSSE with forward and backward privacy for conjunctive queries (named ODXT) by deploying the framework of OXT [9], where it exists two encrypted datasets (named "TSet" and "XSet"). The former is used to get files matching the least frequent keyword in a conjunctive query, and the latter is used to test whether the matching files contain the remaining keywords in the conjunctive queries.



(a) In OXDT, the server can repeat the query issued at time t_1 after t_2 and get ind_1. This is due to the fact that ind_1 with w_2 is recovered in the "XSet" with the search query happened at time t_1, where the "XSet" is not forward private. As a result, ODXT is not forward private in this circumstance since the server learns the record inserted at t_2 matched the previous search query.



(b) At the time t_2, (w_2, ind_1) is deleted by inserting a new record with an indicator "del", which is called "lazy deletion". However, for the query at t_3, the server still needs to check the 2 records related to w_2, which increases the communication and interactions between the client and the server.

Figure 1: Illustration of conjunctive queries.

Unfortunately, ODXT [49]'s security is not comprehensive enough for conjunctive queries with OXT framework. In particular, only the forward privacy of "TSet" is guaranteed, but not for the "XSet". For example (see Fig. 1(a)), given the following update and search queries: $\{t_0, add, (w_1, ind_1)\}$, $\{t_1, search, (w_1, w_2)\}$[2], and $\{t_2, add, (w_2, ind_1)\}$. For the search query that happened at time t_1 (assume w_1 is the least frequent keyword), the server first searches w_1 (in the "TSet") and gets ind_1. Then the server tests if ind_1 contains w_2 in the "XSet", and the final result is that there are no matching files for the query (w_1, w_2). After the update query happened at time t_2, the pair (w_2, ind_1) has been added to the "XSet". Then, according to ODXT, the server still can use the previously issued search query (happened at time t_1) to search the updated "XSet" and get the final result ind_1. However, forward privacy requires that a server cannot map newly added files (e.g., ind_1) to previously issued search queries (e.g., $\{t_1, search, (w_1, w_2)\}$).

[2] (w_1, w_2) denotes a conjunctive query for keywords w_1 and w_2.

In addition, many DSSE schemes deploy the "lazy deletion" [6, 49], where the deletion is achieved by insertion with an indicator "del". For example, as demonstrated in Fig. 1(b), assume there are following update and search queries: $\{t_0, add, (w_1, ind_1)\}$, $\{t_1, add, (w_2, ind_1)\}$, $\{t_2, del, (w_2, ind_1)\}$, and $\{t_3, search, (w_1, w_2)\}$. For the search query that happened at time t_3, ind_1 will not be returned, yet the server still needs to store the encrypted ind_1 for both addition and deletion, which incurs more storage cost. Keeping deleted items as records also requires more communication and interactions between the client and the server, which greatly degrade the efficiency of the DSSE schemes. We need more efficient deletions (e.g., minimize the number of interactions).

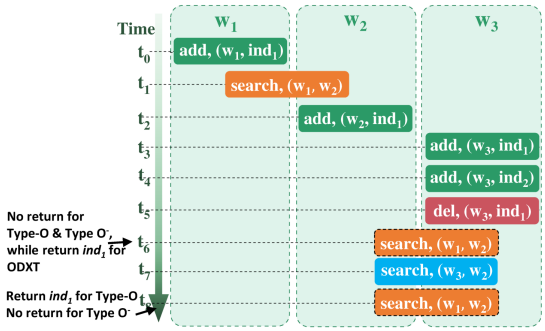


Figure 2: Illustration of Type-O and Type-O$^-$. In a Type-O scheme, the server gets nothing when repeating the first query (t_1) at t_6, but can get ind_1 at t_8 by reusing certain data from the query issued at t_7. In Type-O$^-$, the server learns nothing at t_6 and t_8. Note that, for ODXT, the server can get ind_1 at t_6 when reusing the first query (t_1).

Our Contributions. To address the aforementioned problems, in this paper, we introduce two new schemes (named SDSSE-CQ and SDSSE-CQ-S), where our schemes guarantee the forward and backward privacy of both "TSet" and "XSet" by carefully applying Aura [54] with non-interactive deletion. Table 1 compares our results with the related works. The concrete contributions of this work are as follows:

- We point out that the forward privacy of the state-of-the-art is not comprehensive. To address this problem, we first give a new DSSE with conjunctive queries (named SDSSE-CQ) based on the framework of OXT [9], where the forward privacy of both "TSet" and "XSet" is guaranteed. Moreover, to reduce the deletion interactions, we apply a DSSE with forward and backward privacy (termed Aura) from [54][3], which achieves non-interactive deletion and the server can filter out deleted entries in both "TSet" and "XSet". To further reduce the leakages of SDSSE-CQ, we introduce another scheme (named SDSSE-CQ-S) by increasing the security of the "XSet" at the cost of degraded efficiency. More details can be found in Section 3.

[3] Note that Aura only supports single keyword queries, and it can be replaced with any DSSE with forward and backward privacy for single keyword queries, e.g., the OSSE proposed in [11].

Table 1: Comparison with previous work

Scheme	Client Storage	Communication		Computation		Non-interactive	FP	BP	Query
		Search	Update	Search	Update	Deletion			Type
IM-DSSE [32]	$O(\|\mathbf{W}\| + D)$	$O(D)$	$O(\|\mathbf{W}\|)$	$O(D)$	$O(\|\mathbf{W}\|)$	✓	✓	Type-I	Single
Aura [54]	$O(\|\mathbf{W}\|d)$	$O(n_w - d_w)$	$O(1)$	$O(n_w)$	$O(1)$	✓	✓	Type-II	Single
ODXT [49]	$O(\|\mathbf{W}\| \log D)$	$O(n_q + d_q)$	$O(1)$	$O(n_q + d_q)$	$O(1)$	✗	✓†	Type-II‡	Conjunctive
SDSSE-CQ	$O(\|\mathbf{W}\|d)$	$O(n_q - d_q)$	$O(1)$	$O(n_q)$	$O(1)$	✓	✓	Type-O	Conjunctive
SDSSE-CQ-S	$O(\|\mathbf{W}\|d)$	$O(n_q - d_q)$	$O(1)$	$O(n_q)$	$O(1)$	✓	✓	Type-O$^-$	Conjunctive

$|\mathbf{W}|$ denotes number of keywords in a database. D and d are the number of files in a database and the length of each entry for a keyword, respectively (note that d is slightly longer than $\log D$). n_w (resp., n_q) and d_w (d_q) are addition and deletion numbers for a keyword w (resp., a conjunctive query q). **FP**, **BP**, Single and Conjunctive stand for Forward Privacy, Backward Privacy, Single keyword queries, and Conjunctive queries, respectively. † The forward privacy of ODXT is not comprehensive for conjunctive queries. ‡ Type-II is not suitable for conjunctive queries. Type-O and Type-O$^-$ are for conjunctive queries with OXT framework, and Type-O$^-$ is stronger than Type-O.

- To precisely quantify the leakages of our proposed schemes, we introduce two new levels of backward privacy (named Type-O and Type-O$^-$, where Type-O$^-$ leaks less leakage than Type-O), which can fully describe the leakages for conjunctive queries with the OXT [9] framework. Type-O$^-$ is usually harder to achieve and requires more computation time than the scheme with Type-O. Informally, Type-O means both "TSet" and "XSet" are individually backward private. That is, after updating either "TSet" or "XSet" without processing new queries, the server cannot link the updates (either add or delete) with previous queries. However, the server could do that after a new search. Type-O$^-$ addresses the issue. The key difference between Type-O and Type-O$^-$ is that whether the newly searched "xterm"s (e.g., w_2 from $(t_7, search, (w_3, w_2))$ in Fig. 2) can be reused when repeating previously issued queries with the same "xterm". Note that other cases (e.g., two queries with the same least frequent keyword) are already considered in Type-O. Each keyword is protected by a DSSE with forward and backward privacy for single keyword queries. In other words, if we consider single keyword queries only, then Type-O and Type-O$^-$ will be degraded to Type-I/II/III[4]. See Section 4 for details.
- Eventually, the experimental evaluation of our schemes and ODXT is given. Compared with ODXT, our proposed schemes achieve better security with reasonable computation and communication increase. See Section 5 for details.

1.1 Related Work

Song et al. [50] first addressed keyword search over encrypted data by deploying symmetric encryption, which is known as searchable symmetric encryption (SSE). However, it is less efficient since every keyword in each file needs to be tested (searched). To improve the search efficiency, Goh [24] proposed a scheme with secure indexes, where each file only needs to be tested once. To further improve the search efficiency, Curtmola et al. [14] gave a sublinear search time SSE by deploying an inverted index data structure. They also formalized the SSE security (i.e., Real vs. Ideal), which has been adopted by the following works. Later, many SSE schemes with different improvements have been introduced (e.g., rich queries [9, 18, 20, 41, 61], dynamism [8, 36], multi-client model [14, 53], locality [4, 10, 17, 43, 44], small client storage [16], etc.).

To make SSE support the update of the encrypted database, dynamic SSE (DSSE) schemes [8, 36] have been introduced. However, these schemes leak extra information during updates. These can be abused by adversaries [2, 7, 25, 26, 59], which highlights the importance of forward and backward privacy. They are informally introduced by Stefanov et al. [51]. Bost [5] has formally defined forward privacy, and the formal backward privacy is formalized by Bost et al. [6]. In particular, they introduced Type-I, Type-II, and Type-III backward privacy, where Type-I is the strongest and Type-III is the weakest. In addition, they gave several DSSE schemes with varied backward privacy. Specifically, the Type-I backward private (called MONETA) shows the feasibility of the Type-I backward private DSSE, which is relied on the TWORAM [23]. Furthermore, They proposed FIDES with Type-II backward privacy. DIANA$_{del}$ and Janus, achieve Type-III backward privacy, are more efficient.

To further improve the efficiency of Janus, Sun et al. [55] introduced a Type-III backward private DSSE (Janus++), which deploys their proposed Symmetric Puncturable Encryption (SPE). Concurrently, Chamani et al. [12] introduced a DSSE (MITRA) with forward and Type-II backward privacy, while it needs to generate search tokens for each entry. To reduce the search tokens, they also deployed the Path ORAM [52] to construct DSSE schemes with forward and backward privacy (ORION and HORUS). Zuo et al. [62] introduced FB-DSSE by using the bitmap index and simple symmetric encryption with homomorphic addition, which achieves Type-I$^-$ backward privacy. In particular, Type-I$^-$ does not include the insertion time of documents containing the search keyword, while Type-I does. Recently, Sun et al. [54] introduced a non-interactive DSSE with forward and Type-II backward privacy (named Aura).

A recent work [32] proposed to use an encrypted incidence matrix to ensure efficient searches and updates with backward

[4]It depends on which level of backward privacy the underlying DSSE achieves. For example, we deploy Aura, and then they become Type-II backward privacy.

privacy guarantees. With the matrix, search and update will be transformed to a linear scan over one matrix row and column, respectively. As shown in Table 1, although this method results in higher asymptotic complexity and is unable to efficiently support non-interactive updates, it can noticeably reduce the update leakage, achieving a higher backward privacy level than Type-I. Besides, the real-world runtime cost of a matrix-based construction can be reduced by avoiding the use of asymmetric primitives.

However, the aforementioned DSSE schemes support single keyword queries only. To address the problem, Kamara et al. [34] presented a dynamic forward private DSSE for boolean queries (named DIEX), which needs a forward private muti-map. However, they did provide a concrete forward private muti-map instance. In addition, it is not backward private. Later, Zuo et al. [61] gave two forward/backward DSSE schemes with range queries (named SchemeA and SchemeB), where the first scheme is forward private, and the other one is backward private. Wang et al. [56] proposed a generic DSSE with forward privacy for range queries based on SchemeA. In addition, they deployed the "lazy deletion" technique from [6] to make the scheme support backward privacy, which is less efficient. After that, Zuo et al. [63] introduced FBDSSE-RQ, which applies the framework of FB-DSSE [62]. Recently, Patranabis et al. [49] introduced a DSSE with forward and backward privacy for conjunctive queries (ODXT), which is based on the framework of OXT [9]. However, as mentioned before, its security is not comprehensive for conjunctive queries. Guo et al. [28] proposed a forward private verifiable DSSE for conjunctive queries, while its backward privacy essentially corresponds to single keyword queries. It is not an easy task to give full forward and backward private DSSE for conjunctive queries because we need to quantify both the leakages of each keyword in a conjunctive query and the keyword conjunction leakages, which is complicated.

There is also a line of researches that design efficient DSSE schemes [21, 22, 29, 30] based on secure hardware (e.g., Intel SGX [60]). In particular, Hoang et al. [30] use SGX to propose an oblivious search and update DSSE on a very large dataset. It is much more efficient than other ORAM-based DSSE while maintaining the secure search and update of the traditional ORAM scheme. To make the oblivious DSSE work in the multi-user setting, many schemes deploy a trusted proxy to execute an ORAM protocol over the network, which incurs more communication cost. Thus, it requires a high bandwidth network with low latency. To reduce the network influence of proxy-based designs, Hoang et al. [29] proposed an oblivious DSSE in the multi-user setting by utilizing the secure enclaves of the SGX to execute the proxy logic.

Another genre of DSSE deploys multiple servers (i.e., distributed setting) to achieve forward and backward privacy [13, 15, 31]. For example, Hoang et al. [31] proposed a more efficient oblivious distributed DSSE based on multiple servers by utilizing the composition of Private Information Retrieval (PIR) and generic ORAM. Later, Dauterman et al. [15] proposed DORY by applying a different technique (i.e., Distributed Point Function (DPF)) to multiple servers, which is efficient. Recently, Chen et al. [13] proposed a more secure file-sharing system (named Titanium) by deploying circuit ORAM, which achieves confidentiality and integrity against malicious servers and users.

Alongside the development of SSE, the leakage-abuse attacks on them also have a long research line. The majority of existing attacks primarily focus on single-keyword search [2, 7, 27, 33, 45–47, 57–59] and range queries [39, 40, 42], and they mainly exploit several types of leakage, such as the database's statistics, access pattern, search pattern, and volume of results. Those attacks also work on OXT-based conjunctive SSEs, which usually leak more leakage such as the co-occurrence of keywords in documents. Based on the adversary's behavior, the attacks are either passive or active. In passive ones, such as [2, 33, 45, 47], the attackers just observe and analyze the results of the client's queries based on some auxiliary information. The active ones [58, 59] relax the requirement of auxiliary information by allowing attackers to maliciously inject or delete records from the databases and observing how they affect the search results. Achieving forward and backward privacy is necessary to prevent the active ones.

2 Preliminaries

In this section, we introduce the necessary cryptographic primitives and the complexity assumption. || denotes the concatenation of two strings, $|\mathbf{S}|$ denotes the cardinality of the set $\mathbf{S}$, and λ denotes the security parameter.

2.1 Decisional Diffie-Hellman (DDH) Assumption

Let $a, b, c \in \mathbb{Z}_p^*$ and g be a generic generator of cyclic group $\mathbb{G}$ of order $p = p(\lambda)$. We say that DDH assumption holds in $\mathbb{G}$ if the advantage $\mathbf{Adv}_{\mathcal{A}}^{\mathsf{DDH}}(\lambda)$ is negligible for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ to distinguish the tuple (g, g^a, g^b, g^{ab}) from (g, g^a, g^b, g^c). Formally,

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{DDH}}(\lambda) = |\Pr[\mathcal{A}(g, g^a, g^b, g^{ab}) = 1] - \Pr[\mathcal{A}(g, g^a, g^b, g^c) = 1]| \leq negl(\lambda).$$

2.2 Symmetric Encryption

A symmetric encryption (SE) consists of the following polynomial-time algorithms $\mathsf{SE} = (\mathsf{SE{\cdot}Enc}, \mathsf{SE{\cdot}Dec})$:

- $ct \leftarrow \mathsf{SE{\cdot}Enc}(k, m)$: For a secret key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$, the algorithm outputs a ciphertext $ct \in \mathcal{CT}$, where $\mathcal{M}, \mathcal{K}, \mathcal{CT}$ are the message space, key space and ciphertext space, respectively.
- $m \leftarrow \mathsf{SE{\cdot}Dec}(k, ct)$: For the ciphertext ct and the secret key k, this algorithm outputs the message m.

Correctness. For all secret key $k \in \mathcal{K}$, message $m \in \mathcal{M}$ and $ct \leftarrow \mathsf{SE{\cdot}Enc}(k, m)$, it holds that $m = \mathsf{SE{\cdot}Dec}(k, ct)$.

Security. An SE is IND-CPA secure if for every probabilistic polynomial time (PPT) adversary $\mathcal{A}$, its advantage

$$\mathbf{Adv}_{\mathsf{SE},\mathcal{A}}^{\mathsf{IND\text{-}CPA}}(\lambda) = |\Pr[\mathcal{A}(\mathsf{SE{\cdot}Enc}(k, m_0)) = 1] - \Pr[\mathcal{A}(\mathcal{A}(\mathsf{SE{\cdot}Enc}(k, m_1)) = 1]|$$

is negligible, where the secret key $k \in \mathcal{K}$ is kept secret, and $\mathcal{A}$ chooses $m_0, m_1 \in \mathcal{M}$ with equal length. In addition, $\mathcal{A}$ can issue more encryption queries adaptively with the restriction that m_0, m_1 are never queried.

2.3 DSSE Definition

We parse a database DB into $\{ind_i, \mathbf{W}_i\}_{i=1}^{D}$, where ind_i is file identifier, $\mathbf{W}_i$ stands for all keywords in file ind_i and file numbers in DB are denoted as D. All different keywords in DB are denoted as $\mathbf{W} = \cup_{i=1}^{D} \mathbf{W}_i$. $N = \sum_{i=1}^{D} |\mathbf{W}_i|$ stands for the number of keyword/identifier pairs in DB. $\mathsf{DB}(q)$ denotes the set of file identifiers matching a search query q.

Now, we briefly introduce the definition of DSSE and its security model (see [63] for details).

DSSE. In this paper, we consider a two-party model, where a client interacts with a server to store and retrieve his/her private data. A DSSE scheme usually contains algorithm **Setup**, **Search**, and **Update**. Specifically, **Setup** is executed by the client. On input λ and DB, it outputs a secret state σ and an encrypted database EDB, where the client keeps the σ secret and the server stores the EDB. **Search** is executed by the client and the server. On input of a search query q, σ and EDB, the client interacts with the server. Eventually, the server outputs nothing and the client outputs files matching q ($\mathsf{DB}(q)$). **Update** is executed by the client and the server. On input $in = (ind, \mathbf{W})$[5], $op \in \{add, del\}$ and σ, this algorithm outputs an updated encrypted database EDB′ to the server and an updated state σ' to the client.

Remark. In (D)SSE literature, two kinds of result models (i.e., result hiding [9] and result revealing [5, 6]) are usually considered. In particular, result hiding indicates that the server sends encrypted results to the client and the client decrypts them. Correspondingly, result revealing means the server decrypts encrypted results and sends plaintexts to the client. We consider the result hiding model throughout this paper. Note that, in our design, only encrypted results are available to the server(see Alg. 1 and 2). However, DSSE's security model usually assumes the server can infer query results(file IDs) from client access patterns because clients eventually access files based on queries. Although oblivious techniques can be used to hide file access patterns with increased computation costs, this is orthogonal to this work.

Common Leakage Functions. Before defining the common leakage functions, we define a conjunctive query $q = (w_1, w_2, \cdots, w_n)$. An update query $u = (op, (w, ind))$, where $op \in \{add, del\}$ is the update operation and (w, ind) denotes a file-identifier/keyword pair. For a search query q, a search pattern leaks the repetition of search queries on each keyword $w \in q$. Formally, $\mathsf{sp}(q) = \{t : \{\mathsf{sp}(w)\}_{w \in q}\}$, where t is a timestamp and $\mathsf{sp}(w) = (t : (t, w))$ leaks the search timestamp of a keyword w. We also define a result pattern $\mathsf{rp}(q) = \{\mathsf{DB}(q)\}$.

Security Model. The security model of DSSE is simulation-based. In particular, there are two worlds (i.e., REAL and IDEAL). REAL follows the original DSSE, while IDEAL is simulated by a simulator $\mathcal{S}$ with the input of defined leakage functions illustrating the leakages leaked during **Setup**, **Search** and **Update** (i.e., $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Search}, \mathcal{L}^{Update})$). If an adversary $\mathcal{A}$ cannot distinguish REAL from IDEAL with overwhelming probability, then the DSSE is $\mathcal{L}$-adaptively-secure. Formally,

Definition 2.1. A DSSE is $\mathcal{L}$-adaptively-secure if for every probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ (with the input $\mathcal{L}$) such that,

$$|\Pr[\mathsf{REAL}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathsf{IDEAL}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq negl(\lambda).$$

2.4 DSSE for Single Keyword Queries

In this subsection, we briefly recall the state-of-the-art forward and Type-II backward private DSSE (named Aura [54]), which supports single keyword queries only. As mentioned before, ODXT deploys "lazy deletion" to achieve backward privacy, which incurs more interactions and is not efficient. To mitigate this, our schemes deploy Aura, where Aura achieves non-interactive deletion[6]. Specifically, Aura consists of the following polynomial-time algorithm $\mathsf{Aura} = (\mathsf{Aura}\cdot\mathbf{Setup}, \mathsf{Aura}\cdot\mathbf{Search}, \mathsf{Aura}\cdot\mathbf{Update})$:

- $(\sigma, \mathsf{EDB}) \leftarrow \mathsf{Aura}\cdot\mathbf{Setup}(1^{\lambda})$: This algorithm runs by the client. For a security parameter λ, it produces an encrypted database EDB and a secret state σ, where σ is kept secret by the client and the EDB is sent to the server.
- $(\sigma', \mathsf{EDB}') \leftarrow \mathsf{Aura}\cdot\mathbf{Update}(op, (w, ind), \sigma; \mathsf{EDB})$: This is a protocol executed between the client and the server. For an operation $op \in \{add, del\}$, a keyword/identifier pair (w, ind), the σ and an encrypted database EDB, it produces an updated EDB′ to the server and an updated σ' to the client.
- $\mathbf{Res} \leftarrow \mathsf{Aura}\cdot\mathbf{Search}(w, \sigma; \mathsf{EDB})$: This is a protocol executed between the client and the server. For a search keyword w, the state σ and the encrypted database EDB, it produces the search result **Res** to the client. Note that this algorithm captures the functionality of the **Search** algorithm, which ideally does not need the server to output information. In fact, the server can learn some information (i.e., leakages) during the interactions between the client and the server, which is depicted in the following sections.

Correctness. Aura deploys a bloom filter, so it inherits the false-positive of the bloom filter. As a result, Aura is *probabilistic correct*, where the false-positive can be negligible by carefully setting the bloom filer. Let Aura be defined as above, it holds that $\Pr[\mathsf{Aura}\cdot\mathbf{Search}(w, \sigma; \mathsf{EDB}) \neq \mathsf{DB}(w)] \leq negl(\lambda)$.

Security. According to [54], we say Aura is forward and Type-II backward private if for every PPT adversary $\mathcal{A}$, its advantage

$$\mathbf{Adv}^{\mathsf{FB}}_{\mathsf{Aura},\mathcal{A}}(\lambda) = |\Pr[\mathsf{REAL}^{\mathsf{Aura}}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathsf{IDEAL}^{\mathsf{Aura}}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]|$$

is negligible (see [54] for details).

We refer readers to Appendix A for detailed Aura protocol.

2.5 Forward Privacy

Informally, forward privacy requires that the server cannot match newly updated files to previously issued queries. In 2016, Bost [5] gave a formal forward privacy definition, which is described below.

Definition 2.2. An $\mathcal{L}$-adaptively-secure DSSE scheme is forward private, if the update leakage function $\mathcal{L}^{Update}$ can be written as

$$\mathcal{L}^{Update}(op, in) = \mathcal{L}'(op, \{(ind_i, \mu_i)\}),$$

[5]Note that, in this paper, we update one keyword/identifier pair for each update. If you want to update more keyword/identifier pairs, you can update many times.

[6]Our schemes can be constructed from any DSSE with forward and backward privacy, which supports single keyword queries.

where ind_i is the updated file, μ_i is the number of keywords corresponding to the updated file ind_i, and $\mathcal{L}'$ is stateless.

In [49], Patranabis et al. deployed the OXT [9] framework to achieve DSSE with forward and backward privacy for conjunctive queries (named ODXT). However, as mentioned before, their security is not comprehensive. This is because OXT contains two data structures (namely "TSet" and "XSet"), and ODXT only guarantees the forward privacy of the "TSet". To address this problem, for each pair of keyword/identifier update, we need to guarantee the forward privacy of both "TSet" and "XSet". To achieve this, we need to define the leakage function for both "TSet" and "XSet". Specifically, the leakage function for update is

$$\mathcal{L}^{Update}(op, w, ind) = \mathcal{L}'((op, ind)_T, (op, ind)_X),$$

where $(op, ind)_T$ and $(op, ind)_X$ are the leakages of (op, ind) for "TSet" and "XSet", respectively.

2.6 Backward Privacy

Informally, backward privacy aims to suppress the information on the updates that the server can learn upon the subsequent query. In this section, we briefly recap the definition of backward privacy and explain why we should reconsider it under the conjunctive query with OXT framework.

Classic backward privacy definition [6] considers a query q with a single keyword w, and depicts the information leaked from all updates (insertion and deletion) regarding w before issuing the query. Bost et al. [6] gave three different levels of backward privacy (namely, Type-III to Type-I, from least secure to most secure) for single keyword queries, which can not describe the leakages of conjunctive queries properly. In particular, we consider the conjunctive queries with the OXT framework, which consists of a keyword "stag" to access "TSet", and a list of keywords "xtag"s to access "XSet".

However, due to the conjunctive query protocol design, we observe that there will be an extra step for matching after querying "TSet", while its backward privacy is not considered by the classic definition. Without loss of generality, with the extra information leakage, if there are two two-keyword conjunctive queries with the same "xtag" (i.e., $q_1 = (w_3, w_2)$ and $q_2 = (w_1, w_2)$, where w_2 is the "xtag") and several updates between q_1 and q_2 for keyword w_2. An adversary can combine the query at q_1 and the information obtained from "XSet" at q_2 to infer the update information regarding w_2 between q_1 and q_2.

For instance, with the following updates and conjunctive queries: $\{t_0, add, (w_3, ind_1)\}, \{t_1, search, (w_3, w_2)\}, \{t_2, add, (w_2, ind_1)\}, \{t_3, add, (w_1, ind_1)\}, \{t_4, add, (w_1, ind_2)\}, \{t_5, del, (w_1, ind_1)\}, \{t_6, search, (w_1, w_2)\}$, a strong backward-private design guarantees that at t_0 and t_6, the query only reveals the number of matched documents, the insertion time, and total number of updates each query keywords, i.e., (w_3, w_2) and (w_1, w_2), when accessing "TSet" and "XSet", respectively. However, with the above-mentioned leakage, after the search query happened at time t_6, the server can learn a subset of "XSet" regarding w_2. The server thus can combine it with previously issued search query $\{t_1, search, (w_3, w_2)\}$ to find that there is one file (i.e., ind_1) contains the keywords w_3 and w_2.

The above example demonstrates that even if all data structures are satisfied with the prior backward privacy definitions, there will be extra update information regarding the query (w_3, w_2) is revealed. To formally describe this information, we define two new levels of backward privacy for conjunctive queries (named Type-O and Type-O$^-$, Type-O$^-$ is more secure). Note that, for each keyword in a conjunctive query, our schemes deploy Aura. Then the leakages of each keyword are the same as the leakages of Aura, which is Type-II backward private. Hence, if there is only one keyword for queries, Type-O/O$^-$ will become Type-II. Specifically,

- Type-O$^-$: For a conjunctive query q, it leaks file identifiers that currently matching q. In addition, it also leaks the number of matching files of each $\{w_1, ws\}_{ws \in W}$ pair, where $q = (w_1, w_2, \cdots, w_n)$[7], W is the collection of any subset of $\{w_2, \cdots, w_n\}$. For each keyword $w \in q$, it leaks the total number of updates and the corresponding update time[8].
- Type-O: Apart from the leakages in Type-O$^-$, it leaks the number of matching files of each $\{w^*, ws\}_{ws \in W}$ pair, where w^* is one of the previously issued least frequent keywords (including the current least frequent keyword w_1) and W is the collection of any subset of $\{w_2, \cdots, w_n\}$.

To formally define the notion, we need to introduce some new leakage functions. For a conjunctive query q, we consider there is an arbitrary number of updates on keywords in q before issuing q, $\mathtt{Time}(q)$ depicts the update time t for each keyword $w \in q$. Formally,

$$\mathtt{Time}(q) = \{t : \{t, op, (w, ind)\}_{w \in q}\}.$$

We let $\mathtt{size}(w_1, W) = \{|\mathtt{DB}(w_1, ws)|\}_{ws \in W}$ denote the number of matching files for any conjunctive queries with form $\{w_1, ws\}_{ws \in W}$, where $q = (w_1, w_2, \cdots, w_n)$, ws is a subset of the conjunctive keywords from $\{w_2, \cdots, w_n\}$, W is the collection of any subset of $\{w_2, \cdots, w_n\}$, and $|\mathtt{DB}(w_1, ws)|$ denotes the number of matching files for the conjunctive query (w, ws) (e.g., (w_1, w_2), (w_1, w_2, w_3))[9]. Similarly, we can define $\mathtt{size}(w^*, W) = \{|\mathtt{DB}(w^*, ws)|\}_{ws \in W}$, where w^* denotes one of the previously issued least frequent keywords (including the current least frequent keyword w_1). For example, for the following update and search queries: $\{t_0, add, (w_3, ind_1)\}$, $\{t_1, search, (w_3, w_2)\}, \{t_2, add, (w_2, ind_1)\}, \{t_3, add, (w_1, ind_1)\}, \{t_4, add, (w_1, ind_2)\}, \{t_5, del, (w_1, ind_1)\}$, and $\{t_6, search, (w_1, w_2)\}$. After time t_6 (assume w_1 and w_3 are the least frequent keywords), we have $\mathtt{size}(w_1, W) = \{|\mathtt{DB}(w_1, w_2)|\} = \{0\}$ and $\mathtt{size}(w^*, W) = \{|\mathtt{DB}(w_1, w_2)|, |\mathtt{DB}(w_3, w_2)|\} = \{0, 1\}$, where W is $\{w_2\}$.

Definition 2.3. An $\mathcal{L}$-adaptively-secure DSSE scheme is Type-O$^-$/O backward private if the update leakage function $\mathcal{L}^{Update}$ and the search leakage function $\mathcal{L}^{Search}$ can be written as the following types, respectively:

- Type-O: $\mathcal{L}^{Update}(op, w, ind) = \mathcal{L}'(op)$ and
  $\mathcal{L}^{Search}(q) = \mathcal{L}''(\mathtt{sp}(q), \mathtt{rp}(q), \mathtt{Time}(q), \mathtt{size}(w^*, W))$,
- Type-O$^-$: $\mathcal{L}^{Update}(op, w, ind) = \mathcal{L}'(op)$ and
  $\mathcal{L}^{Search}(q) = \mathcal{L}''(\mathtt{sp}(q), \mathtt{rp}(q), \mathtt{Time}(q), \mathtt{size}(w_1, W))$,

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

[7] We assume w_1 is the least frequent keyword.
[8] This is inherited from the Type-II backward privacy of Aura. For other DSSE for single keyword queries, it can be changed accordingly (e.g., Type-III).
[9] The number of matching files for each keyword w can be deduced from the $\mathtt{Time}(q)$.

In Section 3, we present two constructions, namely SDSSE-CQ and SDSSE-CQ-S. Our first construction (see Section 3.1) can achieve Type-O backward privacy. To further reduce the leakage, we introduce SDSSE-CQ-S in Section 3.2, which reaches Type-O$^-$ backward privacy. Note that other potential leakage (e.g., two queries with the same least frequent keyword) is already considered in SDSSE-CQ[10].

3 Our Constructions

In this section, we present two DSSE schemes with conjunctive queries. The first scheme (termed SDSSE-CQ) improves the forward privacy of ODXT and supports non-interactive deletion. To further reduce the leakages, we introduce the second DSSE (named SDSSE-CQ-S) with a stronger level of backward privacy.

Overview. Before proceeding, we would like to briefly introduce the high-level ideas of our constructions. OXT deploys two data structures, namely "TSet" and "XSet", which can achieve sublinear search efficiency. In particular, "TSet" is a map that keeps the mapping between the keyword and a list of files that contain the keyword. Meanwhile, "Xset" is a hash list that contains the cryptographic hash values generated with all existing keyword/file ID pairs. When executing a conjunctive query with OXT data structures, the query is executed in two steps. The first step searches the (i.e., least frequent) keyword in the "TSet" through "stag" and gets the matching files. The second step generates an "xtag" with each pair of the file IDs retrieved from the first stage and the remaining keywords in a conjunctive query, and then tests whether the "xtag" is in the "XSet". If the "xtag" is in the "XSet", it means the file retrieved from the first stage also contains other keywords, thus will be included in the conjunctive query result. For detailed construction, we refer readers to Appendix B.

Since OXT does not support update, Patranabis et al. proposed ODXT by applying a forward and backward private DSSE to the OXT data structure [49]. Specifically, it only applies the DSSE scheme to the "TSet" of OXT, while the "XSet" is not forward private. In addition, as mentioned before, ODXT uses "lazy deletion", which incurs more interactions and communication cost. To address these problems, we proposed SDSSE-CQ by carefully integrating the non-interactive DSSE scheme (Aura) to all OXT data structures to ensure their forward and backward privacy while reducing the cost of "lazy deletion".

Furthermore, as we pointed out in Section 2.6, there will be a new leakage under the backward privacy notion when leveraging OXT-like data structures to enable dynamic updates. To further reduce the leakages, we propose SDSSE-CQ-S by carefully adding extra randomness to the "xtag" corresponding to a particular "stag". As a result, the "xtag" of the previous search query cannot be reused by the following search queries, then the search leakages are reduced. More details are depicted in the following sections.

3.1 Our First Scheme

As mentioned before, the security model of ODXT [49] is not comprehensive enough for conjunctive queries with OXT [9] framework. This is due to the fact that, similar to OXT, ODXT has two datasets (named "TSet" and "XSet"). ODXT only guarantees the forward privacy of "TSet", while "XSet" is not forward private. To address the problem, we protect the forward privacy of both "TSet" and "XSet". In addition, for ODXT, the authors deploy the "lazy deletion" to support deletion, which incurs more interactions and is inefficient. To reduce the interactions, we deploy the state-of-the-art construction (Aura, see Section 2.4) from [54], which supports non-interactive deletion and single keyword queries only. Note that, in our construction, Aura can be replaced with any DSSE with forward and backward privacy supporting single keyword queries.

Let Aura = (Aura·**Setup**, Aura·**Search**, Aura·**Update**) be the forward and backward private DSSE from [54]. We give SDSSE-CQ = (SDSSE-CQ·**Setup**, SDSSE-CQ·**Update**, SDSSE-CQ·**Search**) in Algorithm 1, which achieves forward and backward privacy as well as supports conjunctive queries. Without loss of generality, we assume w_1 is the least frequent keyword. The details of the algorithms are described as follows:

- $(\mathsf{EDB}, \sigma) \leftarrow$ SDSSE-CQ·**Setup**(1^λ): A client inputs λ, he/she selects a secret key k for keyed Pseudorandom function (PRF) F and keys k_x, k_i, k_z for keyed PRF F_p (with range in Z_p^*, p is a large prime). In particular, k is used to generate the keys corresponding to each keyword "w" that is used in the "TSet", and k_x, k_i, k_z are used to generate the "xtag"s stored in "XSet". Note that these keys are part of our scheme (rather than Aura) and not used in the **CT** and the EDB setup. Moreover, he/she sets an empty map **CT**, which stores keyword/counter (w/c) pairs. In addition, he/she sets two Aura instances, which are used for "TSet" and "XSet", respectively. Eventually, he/she outputs encrypted database $\mathsf{EDB} = (\mathsf{EDB}_T, \mathsf{EDB}_X)$ to a server and secretly keeps the state $\sigma = (k, k_x, k_i, k_z, \mathbf{CT}, \sigma_T, \sigma_X)$.
- $(\sigma', \mathsf{EDB}') \leftarrow$ SDSSE-CQ·**Update**$(op, w, ind, \sigma; \mathsf{EDB})$: A client inputs an operation op, a keyword w corresponding to a file identifier ind, a state σ and the server inputs the encrypted database EDB. The client generates the encrypted identifier e, y, and $xtag$. Then the client interacts with the server to update $(e||y||c)$ and $xtag$ corresponding to keyword w by using the Aura·**Update** (Alg. 1: line 9-10 of SDSSE-CQ·**Update**). Note that the $(e||y||c)$ and $xtag$ are the secret inputs of the client, which are not revealed to the server before a search query. In addition, $(e||y||c)$ (or $xtag$) acts as the role of "ind" of Aura·**Search**, which is defined in Section 2.4. Eventually, the server outputs an updated EDB EDB′ and the client outputs an updated state σ'.
- $\mathcal{I} \leftarrow$ SDSSE-CQ·**Search**$(q = (w_1, w_2, \cdots, w_n), \sigma; \mathsf{EDB})$: A client inputs a conjunctive query $q = (w_1, w_2, \cdots, w_n)$ and a state σ, and the server inputs EDB. The client first retrieves c from **CT** corresponding to keyword w_1 (Alg. 1: line 1-4 of SDSSE-CQ·**Search**, and we assume w_1 is the least frequent keyword). Then he/she generates the $xtoken$ for each i from 0 to c and j from 2 to n, which will be sent to the server (Alg. 1: line 5-8 of SDSSE-CQ·**Search**). After that, the client interacts with the server to search w_1 (gets the current "xtag"s for keywords $(w_2, \cdots, w_n)$) from EDB_T (EDB_X) through Aura·**Search** (Alg. 1: line 9-20 of SDSSE-CQ·**Search**). Note that, after a search query, the update time of Aura is

[10]Forward and backward privacy do not protect volume information, while some prior works [35, 48] can achieve the volume-hiding feature, we mark those works as orthogonal.

Algorithm 1 SDSSE-CQ

SDSSE-CQ·**Setup**(1^λ)

1: $k \xleftarrow{\$} \{0,1\}^\lambda$ for PRF F (used in "TSet"), $k_x, k_i, k_z \xleftarrow{\$} \{0,1\}^\lambda$ for PRF F_p (with range in Z_p^* and used in "XSet")
2: $\mathbf{CT} \leftarrow$ empty map
3: $(\sigma_T, \mathsf{EDB}_T) \leftarrow$ Aura·**Setup**(1^λ)
4: $(\sigma_X, \mathsf{EDB}_X) \leftarrow$ Aura·**Setup**(1^λ)
5: **return** $(\mathsf{EDB} = (\mathsf{EDB}_T, \mathsf{EDB}_X), \sigma = (k, k_x, k_i, k_z, \mathbf{CT}, \sigma_T, \sigma_X))$

SDSSE-CQ·**Update**$(op, w, ind, \sigma; \mathsf{EDB})$

Client:

1: $c \leftarrow \mathbf{CT}[w]$
2: **if** $c = \perp$ **then**
3: $\quad c \leftarrow -1$
4: **end if**
5: $c \leftarrow c + 1, \mathbf{CT}[w] \leftarrow c$
6: $k_w \leftarrow F(k, w), e \leftarrow \mathsf{SE}\cdot\mathsf{Enc}(k_w, ind)$
7: $xind \leftarrow F_p(k_i, ind), z \leftarrow F_p(k_z, w||c), y \leftarrow xind \cdot z^{-1}$
8: $xtag \leftarrow g^{F_p(k_x, w)\cdot xind}$ ▷ g is the generator of a cyclic group with order p.

Client ↔ Server:

9: Run Aura·**Update**$(op, (w, e||y||c), \sigma_T; \mathsf{EDB}_T)$ ▷ This is a protocol between the client and the server, where $(op, (w, e||y||c), \sigma_T)$ are the inputs of the client and kept secret by the client.
10: Run Aura·**Update**$(op, (w, xtag), \sigma_X; \mathsf{EDB}_X)$
Note that $e||y||c$ (or $xtag$) acts as "ind" of Aura·**Update** defined in Section 2.4.

SDSSE-CQ·**Search**$(q = (w_1, w_2, \cdots, w_n), \sigma; \mathsf{EDB})$

Client:

1: $c \leftarrow \mathbf{CT}[w_1], k_{w_1} \leftarrow F(k, w_1)$
2: **if** $c = \perp$ **then**
3: $\quad$ **return** $\emptyset$
4: **end if**
5: **for** $i = 0$ to c, $j = 2$ to n **do**
6: $\quad xtoken[i, j] \leftarrow g^{F_p(k_z, w_1||i)\cdot F_p(k_x, w_j)}$
7: **end for**
8: Send $xtoken$ to the server.

Client ↔ Server:

9: $\mathbf{Res}_T \leftarrow$ Aura·**Search**$(w_1, \sigma_T; \mathsf{EDB}_T)$ ▷ Getting file identifiers matching the first keyword by searching "TSet".
10: **if** $\mathbf{Res}_T = \perp$ **then**
11: $\quad$ **return** $\emptyset$
12: **end if**
13: $\mathbf{XSet} \leftarrow$ empty set
14: **for** $j = 2$ to n **do**
15: $\quad \mathbf{Res}_X \leftarrow$ Aura·**Search**$(w_j, \sigma_X; \mathsf{EDB}_X)$ ▷ Getting the current "xtag"s corresponding to the remaining keywords.
16: $\quad$ **if** $\mathbf{Res}_X = \perp$ **then**
17: $\quad\quad$ **return** $\emptyset$
18: $\quad$ **end if**
19: $\quad \mathbf{XSet} \leftarrow \mathbf{XSet} \cup \mathbf{Res}_X$
20: **end for**

Server:

21: $\mathbf{Res} \leftarrow$ empty set
22: **for** each $(e||y||c) \in \mathbf{Res}_T$ **do**
23: $\quad flag \leftarrow true$
24: $\quad$ **for** $j = 2$ to n **do**
25: $\quad\quad$ **if** $xtoken[c, j]^y \notin \mathbf{XSet}$ **then** ▷ Testing if the files matching the first keyword contain the remaining keywords through "$xtoken$".
26: $\quad\quad\quad flag \leftarrow false$
27: $\quad\quad$ **end if**
28: $\quad$ **end for**
29: $\quad$ **if** $flag$ **then**
30: $\quad\quad \mathbf{Res} \leftarrow \mathbf{Res} \cup e$
31: $\quad$ **end if**
32: **end for**
33: Send **Res** to the client.

Client:

34: **for** each $e \in \mathbf{Res}$ **do**
35: $\quad ind \leftarrow \mathsf{SE}\cdot\mathsf{Dec}(k_{w_1}, e)$
36: **end for**

leaked. In other words, the counter c is implicitly leaked in Aura after a search query. Hence, the added counter c does not incur a new leakage. The server retrieves all the file identifiers corresponding to w_1 and tests if they contain the keywords $w_2, \cdots, w_n$ with the $xtoken$s. Finally, the server sends all the encrypted file identifiers $\mathcal{I}$ back to the client, and the client can decrypt them (Alg. 1: line 33-36 of SDSSE-CQ·**Search**).

3.2 Stronger Backward Privacy

As mentioned before, SDSSE-CQ achieves Type-O backward privacy. To further reduce the leakage, we introduce Type-O$^-$ at the cost of degraded efficiency, where the newly generated $xtag$s cannot be used by previously issued search queries. To achieve Type-O$^-$ backward privacy, we add a new random number (related to a keyword/counter pair) to an $xtag$. Specifically, we propose SDSSE-CQ-S = (SDSSE-CQ-S·**Setup**, SDSSE-CQ-S·**Update**, SDSSE-CQ-S·**Search**), which is described in Algorithm 2 and the differences from SDSSE-CQ are highlighted in gray.

- $(\mathsf{EDB}, \sigma) \leftarrow$ SDSSE-CQ-S·**Setup**(1^λ): Apart from the keys and maps generated in SDSSE-CQ· **Setup**, it additionally chooses two new secret keys k'_x and k'_z for PRF F_p.
- $(\sigma', \mathsf{EDB}') \leftarrow$ SDSSE-CQ-S·**Update**$(op, w, ind, \sigma; \mathsf{EDB})$: This protocol is almost the same as the SDSSE-CQ·**Update** except that it puts a new randomness (related to the keyword/counter $w||c$) to the $wxtag$ (Alg. 2: line 8 of SDSSE-CQ-S·**Update**), which is used to avoid the newly generated $wxtag$s to be reused by previously issued search queries (with different least frequent keyword).
- $\mathcal{I} \leftarrow$ SDSSE-CQ-S·**Search**$(q = (w_1, w_2, \cdots, w_n), \sigma; \mathsf{EDB})$: This protocol is similar to the **Search** of SDSSE-CQ. The differences are that the client puts the randomness $F_p(k'_z, w_1)$ (corresponding to the least frequent keyword w_1) to the $wxtokens$ ($g^{F_p(k_z, w_1||i)\cdot F_p(k_x, w_j)\cdot F_p(k'_z, w_1)}$) (Alg. 2: line 5-8 of SDSSE-CQ-S·**Search**) and generates new tokens wxt ($F_p(k'_x, w_j||c_j)\cdot F_p(k'_z, w_1)$) (Alg. 2: line 9-17 of SDSSE-CQ-S·**Search**). When the server tries to recover a new "$xtag$" from $wxtag$ ($wxtag^{wxt}$), the randomness $F_p(k'_x, w_j||c_j)$ will be canceled out, and the randomness $F_p(k'_z, w_1)$ (with respect to the w_1) will be added to the new "$xtag$" (Alg. 2: line 29-31

Algorithm 2 SDSSE-CQ-S (Differences from SDSSE-CQ are highlighted in gray)

SDSSE-CQ-S·**Setup**(1^λ)

$k \xleftarrow{\$} \{0,1\}^\lambda$ for PRF F, k_x, k_i, k_z, $k'_x, k'_z \xleftarrow{\$} \{0,1\}^\lambda$ for PRF F_p (with range in Z_p^*)
CT ← empty map
$(\sigma_T, \mathsf{EDB}_T) \leftarrow$ Aura·**Setup**(1^λ)
$(\sigma_X, \mathsf{EDB}_X) \leftarrow$ Aura·**Setup**(1^λ)
return $(\mathsf{EDB} = (\mathsf{EDB}_T, \mathsf{EDB}_X), \sigma = (k, k_x, k_i, k_z, k'_x, k'_z, \mathbf{CT}, \sigma_T, \sigma_X))$

SDSSE-CQ-S·**Update**$(op, w, ind, \sigma; \mathsf{EDB})$

Client:

$c \leftarrow \mathbf{CT}[w]$
if $c = \perp$ **then**
  $c \leftarrow -1$
end if
$c \leftarrow c + 1$, $\mathbf{CT}_T[w] \leftarrow c$
$k_w \leftarrow F(k, w)$, $e \leftarrow \mathsf{SE}\cdot\mathsf{Enc}(k_w, ind)$
$xind \leftarrow F_p(k_i, ind)$, $z \leftarrow F_p(k_z, w||c)$, $y \leftarrow xind \cdot z^{-1}$
$wxtag \leftarrow g^{F_p(k_x,w)\cdot xind \cdot F_p(k'_x, w||c)^{-1}}$

Client ↔ Server:

Run Aura·**Update**$(op, (w, e||y||c), \sigma_T; \mathsf{EDB}_T)$ ▷ This is a protocol between the client and the server, where $(op, (w, e||y||c), \sigma_T)$ are the inputs of the client and kept secret by the client.
Run Aura·**Update**$(op, (w, wxtag||c), \sigma_X; \mathsf{EDB}_X)$

SDSSE-CQ·**Search**$(q = (w_1, w_2, \cdots, w_n), \sigma; \mathsf{EDB})$

Client:

$c \leftarrow \mathbf{CT}[w_1]$, $k_{w_1} \leftarrow F(k, w_1)$
if $c = \perp$ **then**
  return $\emptyset$
end if
for $i = 0$ to c, $j = 2$ to n **do**
  $wxtoken[i, j] \leftarrow g^{F_p(k_z, w_1||i)\cdot F_p(k_x, w_j)\cdot F_p(k'_z, w_1)}$
end for
for $j = 2$ to n **do**
  $c_j \leftarrow \mathbf{CT}[w_j]$
  if $c_j = \perp$ **then**
    return $\emptyset$
  end if
  for $k = 0$ to c_j **do**
    $wxt[j][k] \leftarrow F_p(k'_x, w_j||k) \cdot F_p(k'_z, w_1)$
  end for
end for
Send $(wxtoken, wxt)$ to the server.

Client ↔ Server:

$\mathbf{Res}_T \leftarrow$ Aura·**Search**$(w_1, \sigma_T; \mathsf{EDB}_T)$
if $\mathbf{Res}_T = \perp$ **then**
  return $\emptyset$
end if
WXSet ← empty set
for $j = 2$ to n **do**
  $\mathbf{Res}_X \leftarrow$ Aura·**Search**$(w_j, \sigma_X; \mathsf{EDB}_X)$
  if $\mathbf{Res}_X = \perp$ **then**
    return $\emptyset$
  end if
  for each $wxtag||c_j \in \mathbf{Res}_X$ **do**
    $\mathbf{WXSet} \leftarrow \mathbf{WXSet} \cup wxtag^{wxt[j][c_j]}$
  end for
end for

Server:

Res ← empty set
for each $(e||y||c) \in \mathbf{Res}_T$ **do**
  $flag \leftarrow true$
  for $j = 2$ to n **do**
    if $wxtoken[c, j]^y \notin \mathbf{WXSet}$ **then**
      $flag \leftarrow false$
    end if
  end for
  if $flag$ **then**
    $\mathbf{Res} \leftarrow \mathbf{Res} \cup e$
  end if
end for
Send **Res** to the client.

Client:

for each $e \in \mathbf{Res}$ **do**
  $ind \leftarrow \mathsf{SE}\cdot\mathsf{Dec}(k_{w_1}, e)$
end for

of SDSSE-CQ-S·**Search**). Then these tags can only be used to test the existence of the rest keywords in a conjunctive query $(w_2, \cdots, w_n)$ corresponding to the current least frequent keyword w_1. In other words, they can not be used to test the existence of the rest keywords with a different least frequent keyword. As a result, SDSSE-CQ-S achieves Type-O$^-$ backward privacy.

Remark. Although the aforementioned designs are for a single user, we note that they would be able to support multi-user scenarios with revocation capabilities after slightly editing the protocols. Our insight on the protocols is that the cornerstone of our work is to put "TSet" and "XSet" in OXT into a Type-II forward/backward private data structure and update query design based on the Type-II DSSE scheme to ensure its forward/backward privacy. Therefore, when accessing those new data structures, we still need to generate PRF tokens as in the original OXT. The above observation indicates that we can follow the prior work [37] to control the PRF generation process. Specifically, we can let the data owner distribute the PRF token as (n, θ)-Shamir's shares to n clients. Then, each client should obtain help from $\theta - 1$ clients to issue a correct query. Moreover, the data owner can revoke access with a revocation token that can be used to refresh the database and unrevoked users' shares. The entire process can ensure the security of multi-user SSE design unless the attacker can compromise $\theta - 1$ clients.

4 Security Analysis

Theorem 4.1. *(Adaptive forward and Type-O backward privacy of* SDSSE-CQ*). Let* Aura *be forward and backward private, F, F_p be secure PRFs, DDH assumption holds over $\mathbb{G}$,* SE *be an* IND-CPA *secure symmetric encryption. We define $\mathcal{L}_{\mathsf{SDSSE\text{-}CQ}} = (\mathcal{L}^{Update}_{\mathsf{SDSSE\text{-}CQ}}, \mathcal{L}^{Search}_{\mathsf{SDSSE\text{-}CQ}})$, where $\mathcal{L}^{Update}(op, w, ind) = op$ and $\mathcal{L}^{Search}(q) = (\mathsf{sp}(q), \mathsf{Time}(q), \mathsf{size}(w^*, W))$. Then* SDSSE-CQ *is $\mathcal{L}_{\mathsf{SDSSE\text{-}CQ}}$-adaptively forward and Type-O backward private.*

Proof Sketch: We update the "TSet" and "XSet" by using Aura, hence the forward privacy of SDSSE-CQ is guaranteed by the forward privacy of Aura. The non-interactive deletion is achieved by using the Aura, which is Type-II backward private. To support conjunctive queries, SDSSE-CQ deploys the OXT data structure, which inherits the leakage of the number of matching files for the pair of $\{w^*, ws\}_{ws \in W}$ (w^* is one of the previously issued least frequent keywords and W denotes the collection of any subset of $\{w_2, \cdots, w_n\}$). Hence, SDSSE achieves Type-O backward privacy. Due to the page limitation, we defer the proof to the appendix C.

□

Theorem 4.2. *(Adaptive forward and Type-O$^-$ backward privacy of* SDSSE-CQ-S*). Let* Aura *be forward and backward private, F, F_p be secure PRFs, DDH assumption holds over $\mathbb{G}$,* SE *be an* IND-CPA *secure symmetric encryption. We define $\mathcal{L}_{\text{SDSSE-CQ}} = (\mathcal{L}^{Update}_{\text{SDSSE-CQ}}, \mathcal{L}^{Search}_{\text{SDSSE-CQ}})$, where $\mathcal{L}^{Update}(op, w, ind) = op$ and $\mathcal{L}^{Search}(q) = (\text{sp}(q), \text{Time}(q), \text{size}(w_1, W))$. Then we have* SDSSE-CQ-S *is $\mathcal{L}_{\text{SDSSE-CQ-S}}$-adaptively forward and Type-O$^-$ backward private.*

Proof Sketch: As mentioned before, for SDSSE-CQ, the newly searched *xtag*s still can be used by previously issued search queries (with different least frequent keywords). To avoid this, we introduce SDSSE-CQ-S, which achieves Type-O$^-$. Compared with SDSSE-CQ, SDSSE-CQ-S combines the "*xtag*" with a random value corresponding to the least frequent keyword w_1, which does not influence the forward privacy. The new "*xtag*" (termed as *wxtag*) can only be used to test the remaining keywords corresponding to the current least frequent keyword w_1. Then SDSSE-CQ-S achieves Type-O$^-$ backward privacy. Due to the page limitation, we defer the proof to the appendix C.

□

5 Experimental Evaluation

In this section, we give the experimental evaluation of our proposed schemes and the state-of-the-art ODXT. In addition, we compare our construction with the matrix-based construction IM-DSSE [32].

5.1 Implementation and Settings

The proposed schemes and ODXT are implemented with C++, and we use Aura [54] to construct our schemes to support non-interactive deletion. For Aura, we use the symmetric one-way technique of FB-DSSE from [62] as the underlying forward private DSSE[11]. Similar to [54], the parameters for the bloom filter used in Aura are set as follows: false positive (10^{-4}) and the number of hash functions (5). In addition, the client knows the (w, ind) pair, so we still use this value to generate the tag for compressed symmetric revocable encryption (CSRE, we refer readers to [54] for details). We leverage the group $\mathbb{G}_T$[12] of PBC with the input *a.properties* to enable the elliptical curve-based cryptographic operations (e.g., group multiplication and exponentiation) involved in conjunctive queries.

[11]Note that ODXT deploys the technique of MITRA [12] to achieve forward privacy. However, for a search query, the client needs to generate many tokens for the list of files matching a keyword w and send them to the server.

[12]We do not use the functions related to pairing, but the operations in $\mathbb{G}_T$ (e.g., exponentiation, inversion in the exponent), which are quite efficient in comparison with other groups.

Table 2: The query delay of conjunctive queries with two keywords ($w_1 \wedge w_2$) under ODXT ($|w_2| = 10^7$)

$\lvert w_1 \rvert$	$d = 10$	$d = 10^2$	$d = 10^3$	$d = 10^4$	$d = 10^5$
Delay (ms)	0.1	1	10	100	1000

Table 3: Average addition and deletion time

Scheme	SDSSE-CQ	SDSSE-CQ-S	ODXT	IM-DSSE
Per addition (ms)	0.9	0.9	0.7	0.6
Per deletion (ms)	0.03	0.03	0.7	0.06

We use a testbed with Intel Xeon 2.60GHz and 128 GM RAM to evaluate and compare our design with IM-DSSE and ODXT. For the IM-DSSE, we pick the IM-DSSE$_{\text{I+II}}$ variant for comparison as it is backward-private and has all optimizations presented in [32]. Since IM-DSSE only supports single-keyword queries, we execute its source code in the same testbed and compare its performance with our design over the single-keyword query setting. On the other hand, the source code of ODXT is not publicly available. Hence, we choose to compare our result with the result reported in [49]. Note that the test machine in [49] is the same as ours, but their implementation leverages the multi-threading technology. As shown in Table 2, ODXT only takes 1 s to handle the query with $|w_1| = 10^5$, which is $10 - 20\times$ faster than SDSSE-CQ.

5.2 Evaluation Results

Update Time. We respectively run the addition and deletion of our proposed schemes and ODXT 10,000 times and measure the average running time. The results are presented in Table 3. From Table 3, it can be seen that the addition time of our proposed schemes is tiny (0.9 ms), while it is comparable to the addition time of ODXT (0.7 ms). Although our schemes need to generate the CSRE ciphertexts (see [54] for details) for both "TSet" and "XSet", we use preprocessed pairings to amortize this cost to make it almost the same as in ODXT. For deletion, SDSSE-CQ and SDSSE-CQ-S only need 0.01 ms because they are only required to insert the tag of deleted keyword/identifier pair into a bloom filter, while ODXT uses "lazy deletion", which consumes the same amount of running time as the addition. Note that, although our schemes consume slightly more addition time than ODXT, our schemes achieve better security and incur less deletion time.

On the other hand, IM-DSSE has a similar update cost compared to our proposed schemes, although the theoretical complexity of IM-DSSE is higher. This is because IM-DSSE needs to compute more it relies on a stream cipher fashion to use an AES-CTR to mask the matrix bit-wise instead of encrypting each element in the matrix. Furthermore, the implementation is optimized with parallel processing, mixing the computation process of AES-CTR and the communication process of accessing the server-side matrix to reduce runtime cost [32].

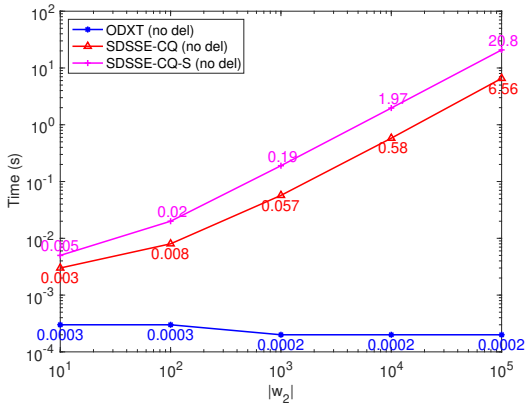


Figure 3: Search time of SDSSE-CQ, SDSSE-CQ-S and ODXT with constant $|w_1|$

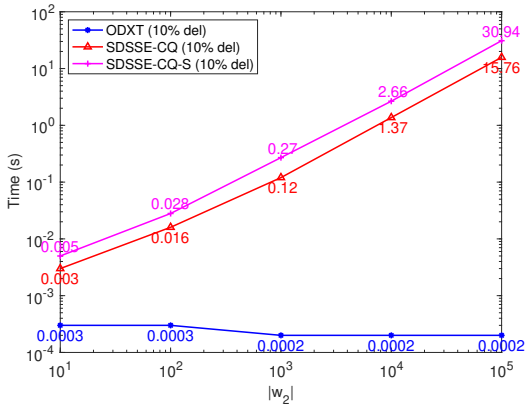


Figure 4: Search time of SDSSE-CQ, SDSSE-CQ-S and ODXT with constant $|w_1|$ and 10% deletion

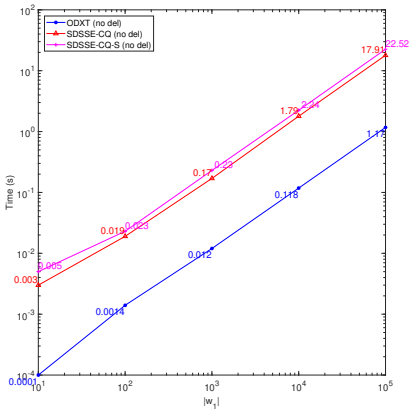


Figure 5: Search time of SDSSE-CQ, SDSSE-CQ-S and ODXT with constant $|w_2|$

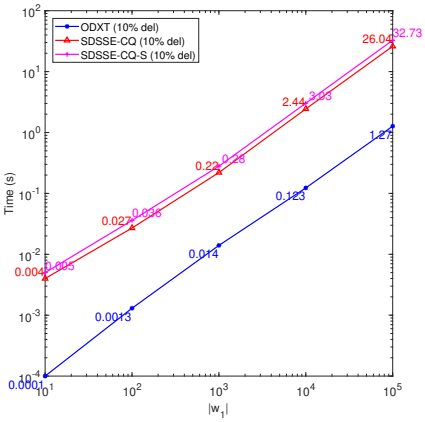


Figure 6: Search time of SDSSE-CQ, SDSSE-CQ-S and ODXT with constant $|w_2|$ and 10% deletion

Search Time. We follow the same setting in [49] to evaluate the schemes. In particular, we execute two types of two-keyword conjunctive queries (w_1, w_2) for the schemes. In the first query type, w_1 has a fixed number of corresponding files (i.e., 10), while the number of files matching w_2 varies from 10 to 100, 000. Correspondingly, the second query type has a fixed number of files (i.e., 10) containing w_2, but the number of files matching keyword w_1 varies from 10 to 100, 000.

The query delay of the first type queries is presented in Fig. 3. It shows that the query can be processed in 0.005 to 20 s. Also, we observe that the query delay increases with the increases of $|w_2|$. This indicates the impact of SRE on our protocol. With the increases of $|w_2|$, our protocols should retrieve more tuples to recover the $xtag$s in the "XSet"`XSet` from the SRE ciphertext. The above operations incur extra costs upon the original OXT protocol. When introducing 10% deletions in the database, more delays can be observed because extra efforts will be made to search the correct keys for decryptions of SRE ciphertexts (see Fig. 4). On the other hand, the search time of ODXT is constant and faster than our schemes because it has a fixed (small) amount of identifiers (i.e., 10) from "TSet" to test. Nonetheless, although the search time of our schemes is longer than the search time of ODXT when $|w_2|$ is increased, our schemes achieve better security. Compared with ODXT, our schemes achieve more comprehensive forward privacy (both "TSet" (e.g., w_1) and "XSet" (e.g., w_2) are forward private, while ODXT only guarantees the forward privacy of "TSet") at the expense of increased search time (i.e., more security operations in "XSet").

A similar trend can be found in Fig. 5 and 6, which demonstrates the search time of the second type of search queries. The difference is that the search time of all schemes is similar and increases with the increase of $|w_1|$. This further demonstrates that the influence of the underlying Aura is dominant in all schemes. In particular, our two test cases retrieved the same amount of items from "TSet" and "XSet", which takes a fixed time to access the underlying EDB. Besides, since we use preprocessed pairing in PBC, the cost of group operations (e.g., exponentiations) is amortized by the preprocessing operation.

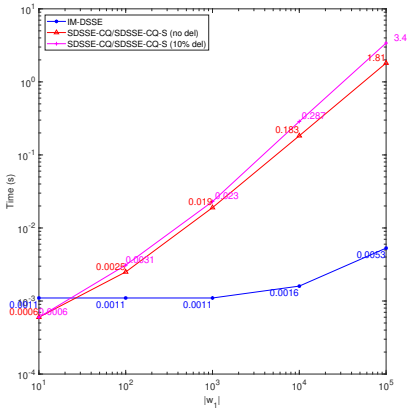


Figure 7: Search time of of SDSSE-CQ, SDSSE-CQ-S and IM-DSSE with single-keyword queries

Table 4: Update communication cost of each keyword/identifier pair

Scheme	SDSSE-CQ	SDSSE-CQ-S	ODXT	IM-DSSE
Addition (Byte)	172	176	156	65
Deletion (Byte)	0	0	156	65

Table 5: Search communication cost for two-keyword conjunctive queries

Scheme	SDSSE-CQ	SDSSE-CQ-S	ODXT
1000	0.13 MB	0.15 MB	0.13 MB
10000	1.23 MB	1.42 MB	1.26 MB
100000	12.21 MB	14.12 MB	12.59 MB

When considering the single keyword queries, both of our proposed protocols will have the same process as the Type-II DSSE scheme Aura we used. As additional operations introduced by OXT are not required, our protocols are much faster when performing single keyword queries (See Fig. 7). Compared to IM-DSSE, although our protocols have a better asymptotic complexity and are symmetric-only for single keyword queries, IM-DSSE is still faster since it is optimized to have more bit operations. We also observe that IM-DSSE shows a constant search time when the number of files ≤ 1000. This is because IM-DSSE encrypts multiple matrix entries in one cipher block, enabling the client to fetch multiple entries at the same time. We believe this will be a potential way to improve our design in future work.

Update Communication Cost. Next, we compare the update communication cost of our proposed schemes with ODXT. We set the key size of the maps, the identifier size, and the counter size to 4 bytes, and the element size in the group $\mathbb{Z}_r$ and $\mathbb{G}_T$ is 20 and 128 bytes, respectively. In addition, we set the operation size of ODXT to 1 byte. As shown in Table 4, the communication cost of SDSSE-CQ-S is slightly larger than the communication cost of SDSSE-CQ, because SDSSE-CQ-S needs to store an additional counter for the "XSet". Thanks to Aura, the deletion of our schemes only involves operations on the client side without the interaction with the server (i.e., non-interaction), it does not incur any communication cost. For ODXT, it has the same communication cost (156 bytes) for both the addition and deletion process, which is slightly smaller than the communication cost of our schemes.

IM-DSSE follows a different update logic, which makes its update communication cost depend on the maximum number of keywords that can be supported by the encrypted database. As a result, in our setting with two keywords, IM-DSSE has a smaller communication cost (only 65 B) for both insertion and deletion. However, with the increasing keyword set size, the update communication cost of IM-DSSE will increase significantly. For instance, with 10 keywords, the communication cost will increase to 2.5 KB. For our construction, the insertion communication cost will be constant if we fix the number of hashes, and the deletion communication cost is always 0 as our protocols are non-interactive.

Search Communication Cost. Table 5 shows the search communication cost of our proposed schemes and ODXT. The search query is a two-keyword conjunctive query (w_1, w_2). The dataset consists of different numbers of files (i.e., 1000, 10,000, and 100,000), where each file contains keywords w_1 and w_2. In addition, we set the key size of the keyed hash functions and AES encryption to 16 bytes. From Table 5, we can see that the search communication cost of all schemes is similar, and it increases with the increase of the number of files. The search communication cost of SDSSE-CQ is slightly smaller than the cost of ODXT. This is due to the fact that ODXT needs to generate an address for each file containing the keyword w_1. The search communication cost of SDSSE-CQ-S is slightly larger than ODXT. This is due to the fact that SDSSE-CQ-S needs to generate additional tokens wxt for files containing keyword w_2. Note that our schemes achieve better security than ODXT.

We further compare our proposed designs with IM-DSSE. Since we only make comparisons under the single keyword setting, the runtime cost of our protocols will be exactly the same as the underlying Type-II DSSE scheme, i.e., Aura [54]. As shown in Table 6, if there is no deletion with our design, it will have a tiny constant communication cost (32 B) to send the keyword trapdoor and root key for the GGM tree, which is much smaller than that in IM-DSSE. On the other hand, if there is a 10% deletion, our design should send the minimal coverage set of the GGM tree to the server, which incurs 37 KB to 362 KB communication costs. We can see that this cost is noticeably larger than IM-DSSE, but is still small in a modern network (e.g., gigabit network). Besides, the asymptotic complexity of our design is better than IM-DSSE (logarithm v.s. linear to the number of files), as we can observe the communication cost difference is decreasing with a larger database.

6 Conclusion

In this paper, we have proposed SDSSE-CQ and SDSSE-CQ-S based on the framework of OXT [9]. In addition, we have given two different levels of backward privacy for conjunctive queries (named

Table 6: Search communication cost for single-keyword queries

Scheme	SDSSE-CQ/SDSSE-CQ-S (No deletion)	SDSSE-CQ/SDSSE-CQ-S (10% deletion)	IM-DSSE
1000	32 B	37 KB	0.13 KB
10000	32 B	205 KB	2 KB
100000	32 B	362 KB	16 KB

Type-O and Type-O$^-$), where Type-O is less secure than Type-O$^-$. Our first scheme (SDSSE-CQ) achieves forward and Type-O backward privacy, and our second scheme (SDSSE-CQ-S) can achieve a stronger level of backward privacy (Type-O$^-$). The security model of our schemes is more comprehensive for conjunctive queries than the ODXT. Moreover, our schemes do not need to send the deleted files to the server, which reduces the interactions.

Acknowledgments

This research is supported by National Key Research and Development Program of China (2023YFB2704000), National Natural Science Foundation of China (62372040), National Natural Science Foundation of China (62272294), National Natural Science Foundation of China (62272413), the ARC Discovery Project (DP200103308), the National Research Foundation, Singapore and Infocomm Media Development Authority under its Trust Tech Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Infocomm Media Development Authority.

References

[1] Ghous Amjad, Seny Kamara, and Tarik Moataz. 2019. Breach-resistant structured encryption. PoPETs 2019, 1 (2019), 245–265.
[2] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2019. Revisiting Leakage Abuse Attacks.. In NDSS 2019. The Internet Society.
[3] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 7 (1970), 422–426.
[4] Angèle Bossuat, Raphael Bost, Pierre-Alain Fouque, Brice Minaud, and Michael Reichle. 2021. SSE and SSD: page-efficient searchable symmetric encryption. In Crypto 2021. Springer, 157–184.
[5] Raphael Bost. 2016. $\Sigma o \varphi o \varsigma$: Forward Secure Searchable Encryption. In CCS 2016. ACM, 1143–1154.
[6] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and backward private searchable encryption from constrained cryptographic primitives. In CCS 2017. ACM, 1465–1482.
[7] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In CCS 2015. ACM, 668–679.
[8] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation.. In NDSS 2014. The Internet Society.
[9] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2013. Highly-scalable searchable symmetric encryption with support for boolean queries. In CRYPTO 2013. Springer, 353–373.
[10] David Cash and Stefano Tessaro. 2014. The locality of searchable symmetric encryption. In EUROCRYPT 2014. Springer, 351–368.
[11] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. 2022. Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions. IACR Cryptol. ePrint Arch. (2022), 648.
[12] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In CCS 2018. ACM, 1038–1055.
[13] Weikeng Chen, Thang Hoang, Jorge Guajardo, and Attila A Yavuz. 2022. Titanium: A metadata-hiding file-sharing system with malicious security. In Network and Distributed System Security (NDSS) Symposium.
[14] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In CCS 2006. ACM, 79–88.
[15] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. 2020. {DORY}: An encrypted search system with distributed trust. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 1101–1119.
[16] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2020. Dynamic Searchable Encryption with Small Client Storage. In NDSS 2020. The Internet Society.
[17] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In Crypto 2018. Springer, 371–406.
[18] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. 2018. Practical private range search in depth. TODS 43, 1 (2018), 2:1–2:52.
[19] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. 2018. Efficient Dynamic Searchable Encryption with Forward Privacy. PoPETs 1 (2018), 5–20.
[20] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. 2015. Rich queries on encrypted data: Beyond exact matches. In ESORICS 2015. Springer, 123–145.
[21] Bernardo Ferreira, Bernardo Portela, Tiago Oliveira, Guilherme Borges, Henrique Domingos, and João Leitão. 2019. BISEN: Efficient Boolean Searchable Symmetric Encryption with Verifiability and Minimal Leakage. In SRDS 2019. 103–10309.
[22] Bernardo Ferreira, Bernardo Portela, Tiago Oliveira, Guilherme Borges, Henrique Domingos, and João Leitão. 2020. Boolean searchable symmetric encryption with filters on trusted hardware. TDSC (2020), 1307–1319.
[23] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In Crypto 2016. Springer, 563–592.
[24] Eu-Jin Goh et al. 2003. Secure indexes. ePrint 2003 (2003), 216.
[25] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In CCS 2018. 315–331.
[26] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2019. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In S&P 2019. IEEE, 1067–1083.
[27] Zichen Gui, Kenneth G. Paterson, and Sikhar Patranabis. 2023. Rethinking Searchable Symmetric Encryption. In 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023. IEEE, 1401–1418.
[28] Cheng Guo, Wenfeng Li, Xinyu Tang, Kim-Kwang Raymond Choo, and Yining Liu. 2023. Forward Private Verifiable Dynamic Searchable Symmetric Encryption with Efficient Conjunctive Query. IEEE Transactions on Dependable and Secure Computing (2023).
[29] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A Yavuz. 2020. MOSE: Practical multi-user oblivious storage via secure enclaves. In Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy. 17–28.
[30] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. 2019. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. Proceedings on Privacy Enhancing Technologies 2019, 1 (2019).
[31] Thang Hoang, Attila A Yavuz, F Betül Durak, and Jorge Guajardo. 2019. A multi-server oblivious dynamic searchable encryption framework. Journal of Computer Security 27, 6 (2019), 649–676.
[32] Thang Hoang, Attila A Yavuz, and Jorge Guajardo. 2019. A secure searchable encryption framework for privacy-critical cloud storage services. IEEE Transactions on Services Computing 14, 6 (2019), 1675–1689.
[33] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012. The Internet Society.
[34] Seny Kamara and Tarik Moataz. 2017. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In Eurocrypt 2017. Springer, 94–124.
[35] Seny Kamara and Tarik Moataz. 2019. Computationally volume-hiding structured encryption. In Eurocrypt 2019. Springer, 183–213.
[36] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In CCS 2012. ACM, 965–976.
[37] Shabnam Kasra Kermanshahi, Joseph K Liu, Ron Steinfeld, Surya Nepal, Shangqi Lai, Randolph Loh, and Cong Zuo. 2019. Multi-Client Cloud-Based Symmetric Searchable Encryption. IEEE Transactions on Dependable and Secure Computing 18, 5 (2019), 2419–2437.
[38] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. 2017. Forward secure dynamic searchable symmetric encryption with efficient

updates. In CCS 2017. 1449–1463.
[39] Evgenios M. Kornaropoulos, Nathaniel Moyer, Charalampos Papamanthou, and Alexandros Psomas. 2022. Leakage Inversion: Towards Quantifying Privacy in Searchable Encryption. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022. ACM, 1829–1842.
[40] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2021. Response-Hiding Encrypted Ranges: Revisiting Security via Parametrized Leakage-Abuse Attacks. In 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. IEEE, 1502–1519.
[41] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shifeng Sun, Dongxi Liu, and Cong Zuo. 2018. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In CCS 2018. ACM, 745–762.
[42] Evangelia Anna Markatou, Francesca Falzon, Roberto Tamassia, and William Schor. 2021. Reconstructing with Less: Leakage Abuse Attacks in Two Dimensions. In CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021. ACM, 2243–2261.
[43] Ian Miers and Payman Mohassel. 2017. IO-DSSE: Scaling Dynamic Searchable Encryption to Millions of Indexes By Improving Locality. In NDSS 2017. The Internet Society.
[44] Brice Minaud and Michael Reichle. 2022. Dynamic Local Searchable Symmetric Encryption. In Crypto 2022. Springer.
[45] Jianting Ning, Xinyi Huang, Geong Sen Poh, Jiaming Yuan, Yingjiu Li, Jian Weng, and Robert H. Deng. 2021. LEAP: Leakage-Abuse Attack on Efficiently Deployable, Efficiently Searchable Encryption with Partially Known Dataset. In CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021. ACM, 2307–2320.
[46] Simon Oya and Florian Kerschbaum. 2021. Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption. In 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. USENIX Association, 127–142.
[47] Simon Oya and Florian Kerschbaum. 2022. IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization. In 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022. USENIX Association, 2407–2424.
[48] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In CCS 2019. 79–93.
[49] Sikhar Patranabis and Debdeep Mukhopadhyay. 2021. Forward and Backward Private Conjunctive Searchable Symmetric Encryption. In NDSS 2021. The Internet Society.
[50] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In S&P 2000. IEEE, 44–55.
[51] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage. In NDSS 2014, Vol. 71. The Internet Society.
[52] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In CCS 2013. ACM, 299–310.
[53] Shi-Feng Sun, Joseph K. Liu, Amin Sakzad, Ron Steinfeld, and Tsz Hon Yuen. 2016. An efficient non-interactive multi-client searchable encryption with support for boolean queries. In ESORICS 2016. Springer, 154–172.
[54] Shi-Feng Sun, Ron Steinfeld, Shangqi Lai, Xingliang Yuan, Amin Sakzad, Joseph K Liu, Surya Nepal, and Dawu Gu. 2021. Practical Non-Interactive Searchable Encryption with Forward and Backward Privacy. In NDSS 2021. The Internet Society.
[55] Shi-Feng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. 2018. Practical Backward-Secure Searchable Encryption from Symmetric Puncturable Encryption. In CCS 2018. ACM, 763–780.
[56] Jiafan Wang and Sherman SM Chow. 2022. Forward and Backward-Secure Range-Searchable Symmetric Encryption. Proceedings on Privacy Enhancing Technologies 1 (2022), 28–48.
[57] Lei Xu, Leqian Zheng, Chengzhi Xu, Xingliang Yuan, and Cong Wang. 2023. Leakage-Abuse Attacks Against Forward and Backward Private Searchable Symmetric Encryption. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023. ACM, 3003–3017.
[58] Xianglong Zhang, Wei Wang, Peng Xu, Laurence T. Yang, and Kaitai Liang. 2023. High Recovery with Fewer Injections: Practical Binary Volumetric Injection Attacks against Dynamic Searchable Encryption. In 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023. USENIX Association, 5953–5970.
[59] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption.. In USENIX Security 2016. 707–720.
[60] Wei Zheng, Ying Wu, Xiaoxue Wu, Chen Feng, Yulei Sui, Xiapu Luo, and Yajin Zhou. 2021. A survey of Intel SGX and its applications. Frontiers of Computer Science 15 (2021), 1–15.
[61] Cong Zuo, Shifeng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. 2019. Dynamic Searchable Symmetric Encryption Schemes Supporting Range Queries with Forward/Backward Privacy. CoRR abs/1905.08561 (2019).
[62] Cong Zuo, Shi-Feng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. 2019. Dynamic Searchable Symmetric Encryption with Forward and Stronger Backward Privacy. In ESORICS 2019. Springer, 283–303.
[63] Cong Zuo, Shi-Feng Sun, Joseph K. Liu, Jun Shao, Josef Pieprzyk, and Lei Xu. 2020. Forward and Backward Private DSSE for Range Queries. TDSC, 328–338.

A Aura

In [54], Sun et al. introduced the non-interactive DSSE scheme Aura. Here, we recall the concrete Aura construction from [54] and refer readers to [54] for more details.

Aura is based on the cryptographic primitive named Compressed Symmetric Revocable Encryption (CSRE), which is also given in [54]. We now present a short description on CSRE, and then show how to construct Aura with CSRE: Let a (b, h, n) bloom filter [3] be $\mathsf{BF} = (\mathsf{BF{\cdot}Gen}, \mathsf{BF{\cdot}Upd}, \mathsf{BF{\cdot}Check})$, and a multi-puncturable PRF be [54] $\mathsf{MF} = (\mathsf{MF{\cdot}Setup}, \mathsf{MF{\cdot}Punc}, \mathsf{MF{\cdot}Eval})$. A Compressed Symmetric Revocable Encryption CSRE is described below.

- $(sk, \mathbf{H}, B) \leftarrow \mathsf{CSRE{\cdot}KGen}(1^\lambda, b, h)$: On input a security parameter λ, and two integers b, h, it runs $\mathsf{BF{\cdot}Gen}(b, h)$ to get $(\mathbf{H}, B)$, where $\mathbf{H} = \{H_j\}_{j\in h}$ and $B = 0^b$. Then it generates a secret key sk by running $\mathsf{MF{\cdot}Setup}(1^\lambda)$.
- $(ct, tag) \leftarrow \mathsf{CSRE{\cdot}Enc}(sk, \mathbf{H}, m, tag)$: On input a secret key sk, a set of hash functions $\mathbf{H}$, and a message m with a tag tag, it generates $i_j \leftarrow H_j(t) \in [b]$, $sk_{i_j} \leftarrow F(sk, i_j)$, and ciphertext $ct_j \leftarrow \mathsf{SE{\cdot}Enc}(sk_{i_j}, m)$, where $j \in [h]$. Then, it outputs $ct = (ct_1, ct_2, \cdots, ct_h)$ together with the tag tag.
- $B_R \leftarrow \mathsf{CSRE{\cdot}Comp}(\mathbf{H}, B, tag)$: On input a set of hash functions $\mathbf{H}$, a bit string B, a tag tag, it outputs a revoked bit string B_R by running $\mathsf{BF{\cdot}Upd}(\mathbf{H}, B, tag)$. In particular, the entries of B indexed by $\{H_j(tag)\}_{j\in[h]}$ are set to 1.
- $sk_R \leftarrow \mathsf{CSRE{\cdot}cKRev}(sk, \mathbf{H}, B_R)$: On input a secret key sk, a set of hash functions $\mathbf{H}$, and a revoked bit string B_R, it first finds a set of indices $I = \{i' \in [b]\}$ from B_R, where $B_R[i'] = 1$. Then it computes the sk_I by running $\mathsf{MF{\cdot}Punc}(sk, I)$ and outputs $sk_R = (sk_I, \mathbf{H}, B_R)$.
- $m/\perp \leftarrow \mathsf{CSRE{\cdot}Dec}(sk_R, ct, tag)$: On input a revoked secret key $sk_R = (sk_I, \mathbf{H}, B_R)$ and a ciphertext $ct = (ct_1, ct_2, \cdots, ct_h)$ with a tag tag, it first checks if $\mathsf{BF{\cdot}Check}(\mathbf{H}, B_R, tag) = 1$. If true, it outputs $\perp$. Otherwise, it finds an index $i* \in [b]$, where $B_R[i*] = 0$ and generates $sk_{i*} \leftarrow \mathsf{MF{\cdot}Eval}(sk_I, i*)$. Finally, it gets the message $m \leftarrow \mathsf{SE{\cdot}Dec}(sk_{i*}, ct_{i*})$.

With CSRE and a forward secure DSSE scheme Σ, we can construct Aura as in Algorithm 3.

B OXT

OXT is an SSE scheme that supports conjunctive keyword queries. In this section, we present a brief overview to help understand the scheme. As mentioned, OXT relies on two data structures, including a "TSet" that keeps the mapping between the keyword and a list of files that contain the keyword, and a "XSet" that contains the cryptographic hash values generated with all existing keyword/file ID pairs. Also, it follows a two-step conjunctive query process, where the first step searches obtain matched files from "TSet" with

Algorithm 3 Aura

Setup(1^λ)

$(\mathsf{EDB}_{\mathsf{add}}, K_{\mathsf{add}}, \sigma_{\mathsf{add}}) \leftarrow \Sigma_{\mathsf{add}}.\mathsf{Setup}(1^\lambda)$
$K_s, K_t \xleftarrow{\$} \{0,1\}^\lambda$, $\mathsf{EDB}_{cache} \leftarrow \emptyset$, **MSK**, **C**, **D** $\leftarrow \perp$
Let $K = (K_{\mathsf{add}}, K_s, K_t)$
return $\big(\sigma = (\sigma_{\mathsf{add}}, \textbf{MSK}, \textbf{C}, \textbf{D}), \mathsf{EDB} = (\mathsf{EDB}_{\mathsf{add}}, \mathsf{EDB}_{cache})\big)$

Update(op, $(w, ind), \sigma$; EDB)

Client:

$msk \leftarrow \textbf{MSK}[w],\ D \leftarrow \textbf{D}[w], i \leftarrow \textbf{C}[w]$
if $msk = \perp$ **then**
  $msk \leftarrow \mathsf{SRE.KGen}(1^\lambda)$, where $msk = (sk, D)$
  $\textbf{MSK}[w] \leftarrow msk,\ \textbf{D}[w] \leftarrow D$
  $i \leftarrow 0,\ \textbf{C}[w] \leftarrow i$
end if
Compute $t \leftarrow F_{K_t}(w, ind)$
if op = add **then**
  $ct \leftarrow \mathsf{SRE.Enc}(msk, ind, t)$
  Run $\Sigma_{\mathsf{add}}.\mathsf{Update}(K_{\mathsf{add}}, \mathsf{add}, w||i, (ct, t), \sigma_{\mathsf{add}}; \mathrm{EDB}_{\mathsf{add}})$
else (*i.e.*, op = del)
  $D \leftarrow \mathsf{SRE.Comp}(D, t)$, $\textbf{D}[w] \leftarrow D$
end if

Search(w, σ; EDB)

Client:

$i \leftarrow \textbf{C}[w],\ (sk, D) \leftarrow \textbf{MSK}[w],\ D \leftarrow \textbf{D}[w]$
if $i = \perp$ **then**
  return $\emptyset$
end if
Compute $sk_R \leftarrow \mathsf{SRE.cKRev}(sk, D)$ ▷ D is from $\textbf{D}[\mathrm{w}]$
Send (sk_R, D) and $tkn = F(K_s, w)$ to server
$msk = (sk, D) \leftarrow \mathsf{SRE.KGen}(1^\lambda)$ ▷ Update msk for w after search
$\textbf{MSK}[w] \leftarrow msk,\ \textbf{D}[w] \leftarrow D, \textbf{C}[w] \leftarrow i + 1$

Client & Server:

Run $\Sigma_{\mathsf{add}}.\mathsf{Search}(K_{\mathsf{add}}, w||i, \sigma_{\mathsf{add}}; \mathsf{EDB}_{\mathsf{add}})$, and server gets a list $((ct_1, t_1), (ct_2, t_2), \ldots, (ct_\ell, t_\ell))$ of ciphertext and tag pairs

Server:

Server uses (sk_R, D) to decrypt all ciphertexts $\{(ct_i, t_i)\}$ as follows
for $i \in [1, \ell]$ **do**
  $ind_i = \mathsf{SRE.Dec}((sk_R, D), ct_i, t_i)$
  if $ind_i \neq \perp$ **then**
    NewInd$\leftarrow$ NewInd $\cup \{(ind_i, t_i)\}$
  else
    DelInd$\leftarrow$ DelInd $\cup \{t_i\}$
  end if
end for
OldInd$\leftarrow$ $\mathrm{EDB}_{cache}[tkn]$
OldInd$\leftarrow$ OldInd$\backslash\{(ind, t) : \exists\ t_i \in \mathrm{DelInd}\ s.t.\ t = t_i\}$
Res$\leftarrow$ NewInd $\cup$ OldInd, $\mathsf{EDB}_{cache}[tkn] \leftarrow$ **Res**
return Res

a keyword trapdoor. The second step generates an "xtag" with each pair of the file IDs retrieved from the first stage and the remaining keywords in a conjunctive query, and then tests whether the "xtag" is in the "XSet". If the "xtag" is in the "XSet", it means the file retrieved from the first stage also contains other keywords, thus will be included in the conjunctive query result. To ensure an efficient check on "XSet", a practical solution is to use a Bloom filter to store cryptographic hash values and enable efficient membership testing.

The detailed algorithm in given in Algorithm 4. Note that OXT is a static SSE scheme, which requires the database input during the Setup phase, and there is no Update algorithm for the original OXT scheme. Nonetheless, new privacy issues (i.e., forward and backward privacy) should be considered when extending OXT to support update functionality.

C Proof of Theorems

Theorem 4.1. *(Adaptive forward and Type-O backward privacy of* SDSSE-CQ*). Let* Aura *be forward and backward private, F, F_p be secure PRFs, DDH assumption holds over $\mathbb{G}$,* SE *be an* IND-CPA *secure symmetric encryption. We define $\mathcal{L}_{\mathsf{SDSSE\text{-}CQ}} = (\mathcal{L}^{Update}_{\mathsf{SDSSE\text{-}CQ}}, \mathcal{L}^{Search}_{\mathsf{SDSSE\text{-}CQ}})$, where $\mathcal{L}^{Update}(op, w, ind) = op$ and $\mathcal{L}^{Search}(q) = (\mathsf{sp}(q), \mathsf{Time}(q), \mathsf{size}(w^*, W))$. Then* SDSSE-CQ *is $\mathcal{L}_{\mathsf{SDSSE\text{-}CQ}}$-adaptively forward and Type-O backward private.*

Proof. The proof consists of a series of games from REAL to IDEAL, and we argue that the adversary $\mathcal{A}$ cannot distinguish between any two consecutive games.

Game G_0: The game is exactly the same as the original DSSE scheme (see Algorithm 1). Then we have

$$\Pr[\mathsf{REAL}^{\mathsf{SDSSE\text{-}CQ}}_{\mathcal{A}}(\lambda) = 1] = \Pr[G_0 = 1].$$

Game G_1: In this game we replace the keyed PRFs F (resp., F_p with k_x, k_i, k_z) with a truly random function[13]. For a new keyword w (resp., w, ind, $w||c$), they choose new values and store them in table Key (resp., G_x, G_i, G_z). For a queried keyword, we retrieve the values from the corresponding tables. Then we can establish an adversary $\mathcal{B}_1$ to distinguish the keyed PRF from a truly random function if an adversary $\mathcal{A}$ can distinguish G_1 from G_0. So we have

$$\Pr[G_0 = 1] - \Pr[G_1 = 1] \leq 4\mathbf{Adv}^{\mathsf{prf}}_{F,\mathcal{B}_1}(\lambda).$$

Game G_2: In this game, similar to [9], we choose a random value r from Z_p and generate the corresponding $xtag \leftarrow g^r$ for the "XSet". In addition, we store the values in the set XTag. If an adversary $\mathcal{A}$ can distinguish G_2 from G_1, then we can build an adversary $\mathcal{B}_2$ to break the DDH assumption. So

$$\Pr[G_1 = 1] - \Pr[G_2 = 1] \leq \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{B}_2}(\lambda).$$

Game G_3: This game is similar to G_2 except that we encrypt a constant 0 by using the symmetric encryption SE. If an adversary $\mathcal{A}$ can distinguish G_3 from G_2, then we can establish an adversary $\mathcal{B}_3$ to break the IND-CPA security of the standard symmetric key encryption SE. So

$$\Pr[G_2 = 1] - \Pr[G_3 = 1] \leq \mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{SE},\mathcal{B}_3}(\lambda).$$

Game G_4: In this game, we can use the leakage of $\mathsf{Time}(q)$ and $\mathsf{size}(w^*, W)$ to choose and set the random values for $xtokens$ properly. This has no influence to the distribution of G_4, Then

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

Game G_5: For the update and search of Aura [54], we can use the same technique to simulate the corresponding values by using the leakages defined in $\mathcal{L}_{\mathsf{SDSSE\text{-}CQ}}$. If the adversary $\mathcal{A}$ can distinguish

[13] We replaced four keyed PRFs.

Algorithm 4 OXT

Setup(1^λ, DB)

Initialise TSet $T \leftarrow \emptyset$ indexed by keywords $\mathbf{W}$.
$K_S, K_I, K_Z, K_X \xleftarrow{\$} \{0,1\}^\lambda$.
Run $\mathsf{BF}\cdot\mathsf{Gen}(m, h)$ to get $(\mathbf{H}, B)$, where $\mathbf{H} = \{H_j\}_{j\in h}$ and $B = 0^m$ to keep XSet
for $w \in \mathbf{W}$ **do**
  Initialise $\mathbf{t} \leftarrow \{\}$.
  Compute $K_e \leftarrow F(K_S, w)$.
  for $ind \in \mathsf{DB}(w)$ **do**
    Set a counter $c \leftarrow 1$.
    Compute $xid \leftarrow F(K_I, ind)$.
    Compute $z_w \leftarrow F(K_Z, w||c)$; $y_c \leftarrow xid \cdot z_w^{-1}$.
    Compute $e_c \leftarrow \mathsf{Sym.Enc}(K_e, ind)$.
    Append (y_c, e_c) to $\mathbf{t}$ and set $c \leftarrow c + 1$.
  end for
  Set $\mathbf{T}[w] \leftarrow \mathbf{t}$.
end for
for $w \in \mathbf{W}$ **do**
  for $ind \in \mathsf{DB}(w)$ **do**
    Compute $xid \leftarrow F_p(K_I, ind)$.
    Compute $xtag \leftarrow g^{F_p(K_X, w)\cdot xid}$
    Run $\mathsf{BF}\cdot\mathsf{Upd}(\mathbf{H}, B, xtag)$
  end for
end for
return $\mathbf{H}$, $mk = (K_S, K_I, K_Z, K_X, K_T)$, $\mathsf{EDB} = (T, B)$.

Search($\mathbf{H}$, mk, $q = (w_1 \wedge \cdots \wedge w_n)$, EDB)

Client & Server:

Client computes $stag \leftarrow F(K_T, w_1)$ and sends $stag$ to the server.
Server Gets T from EDB.
Server retrieves $t \leftarrow T[stag]$, sends $|t|$ to client, and starts accepting $xtoken$s computed by client as follows:
for $c = 1 : |\mathbf{t}|$ **do**
  Client computes $\eta_{w_1} \leftarrow F_p(K_Z, w_1||c)$.
  for $\ell = 2 : n$ **do**
    Client computes $xtoken[c, \ell] \leftarrow g^{\eta_{w_1}\cdot F_p(K_X, w_\ell)}$.
  end for
  Client sets $xtoken[c] \leftarrow (xtoken[c, 2], \ldots, xtoken[c, n])$.
  Client sends $xtoken[c]$ to server.
end for
Gets B from EDB.
Server initialises $\mathcal{E} \leftarrow \{\}$.
for $c = 1 : |t|$ **do**
  Server recovers (y_c, e_c) from the c-th component of $\mathbf{t}$.
  for $\ell = 2 : n$ **do**
    Server computes $xtag = xtoken[c, \ell]^{y_c}$.
    if $\mathsf{BF}\cdot\mathsf{Check}(\mathbf{H}, B, xtag) = 0$ **then**
      break
    end if
    Add e_c into $\mathcal{E}$
  end for
end for
Server sends $\mathcal{E}$ to client.
Client computes $K_e \leftarrow F(K_S, w_1)$,
Client computes $ind_c \leftarrow \mathsf{Sym.Dec}(K_e, e_c)$, and adds ind_c to **Res** for all $e_c \in \mathcal{E}$.
return Res

G_5 from G_4, then we can build an adversary $\mathcal{B}_4$ to break the forward and backward privacy of Aura. Therefore

$$\Pr[G_4 = 1] - \Pr[G_5 = 1] \leq \mathbf{Adv}^{\mathsf{FB}}_{\mathsf{Aura},\mathcal{B}_4}(\lambda).$$

Simulator: $\mathsf{sp}(q) = min\{\mathsf{sp}(w)\}_{w\in q}$ can be used to simulate the search queries. Moreover, the construction of encrypted database EDB can be properly simulated by using the leakage of $\mathsf{Time}(q)$ and $\mathsf{size}(w_1 \wedge W)$ as the input of SDSSE-CQ·**Search**. Then we can find that G_5 can be simulated by the simulator $\mathcal{S}$ with the defined leakages. Then we have

$$\Pr[G_5 = 1] = \Pr[\mathtt{IDEAL}^{\mathsf{SDSSE\text{-}CQ}}_{\mathcal{A},\mathcal{S}} = 1].$$

Eventually, any adversary $\mathcal{A}$ attacking SDSSE-CQ is

$$\Pr[\mathtt{REAL}^{\mathsf{SDSSE\text{-}CQ}}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathtt{IDEAL}^{\mathsf{SDSSE\text{-}CQ}}_{\mathcal{A},\mathcal{S}} = 1] \leq$$

$$4\mathbf{Adv}^{\mathsf{prf}}_{F,\mathcal{B}_1}(\lambda) + \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{B}_2}(\lambda) + \mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{SE},\mathcal{B}_3}(\lambda) + \mathbf{Adv}^{\mathsf{FB}}_{\mathsf{Aura},\mathcal{B}_4}(\lambda).$$

□

Theorem 4.2. *(Adaptive forward and Type-O$^-$ backward privacy of SDSSE-CQ-S). Let Aura be forward and backward private, F, F_p be secure PRFs, DDH assumption holds over $\mathbb{G}$, SE be an IND-CPA secure symmetric encryption. We define $\mathcal{L}_{\mathsf{SDSSE\text{-}CQ}} = (\mathcal{L}^{Update}_{\mathsf{SDSSE\text{-}CQ}}, \mathcal{L}^{Search}_{\mathsf{SDSSE\text{-}CQ}})$, where $\mathcal{L}^{Update}(op, w, ind) = op$ and $\mathcal{L}^{Search}(q) = (\mathsf{sp}(q), \mathsf{Time}(q), \mathsf{size}(w_1, W))$. Then we have SDSSE-CQ-S is $\mathcal{L}_{\mathsf{SDSSE\text{-}CQ\text{-}S}}$-adaptively forward and Type-O$^-$ backward private.*

Proof. Similar to the proof of Theorem 1, we can set a series of games from $\mathtt{REAL}^{\mathsf{SDSSE\text{-}CQ\text{-}S}}_{\mathcal{A}}(\lambda)$ to $\mathtt{IDEAL}^{\mathsf{SDSSE\text{-}CQ\text{-}S}}_{\mathcal{A},\mathcal{S}}(\lambda)$ and proof that every two consecutive games are indistinguishable.

The difference is that, in G_1, we need to additionally use two truly random functions to replace F_p with two new keys k'_x, k'_z. Then we can simulate the $wxtag$, $wxtoken$, and wxt[14]. As a result, we can conclude that the advantage of any adversary $\mathcal{A}$ attacking SDSSE-CQ-S is

$$\Pr[\mathtt{REAL}^{\mathsf{SDSSE\text{-}CQ\text{-}S}}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathtt{IDEAL}^{\mathsf{SDSSE\text{-}CQ\text{-}S}}_{\mathcal{A},\mathcal{S}} = 1] \leq$$

$$6\mathbf{Adv}^{\mathsf{prf}}_{F,\mathcal{B}_1}(\lambda) + \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{B}_2}(\lambda) + \mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{SE},\mathcal{B}_3}(\lambda) + \mathbf{Adv}^{\mathsf{FB}}_{\mathsf{Aura},\mathcal{B}_4}(\lambda).$$

□

Remark. The above proofs of the theorems have shown that our proposed schemes are secure under the Real-Ideal world (i.e., proved Definition 2.1), and thus are $\mathcal{L}_{\mathsf{SDSSE\text{-}CQ}}$ (or $\mathcal{L}_{\mathsf{SDSSE\text{-}CQ\text{-}S}}$)-adaptively secure schemes. In addition, forward privacy requires the update not to leak keyword information (see Definition 2.2). In the security proofs of the above theorems, the update does not leak this information, which implies forward privacy (i.e., proved Definition 2.2).

[14]We replaced six keyed PRFs.