Searchable Encryption for Conjunctive Queries with Extended Forward and Backward Privacy

Cong Zuo[1], Shangqi Lai[2], Xingliang Yuan[2], Joseph K. Liu[2], Jun Shao[3,*], and Huaxiong Wang[1]

[1] SCRIPTS, Nanyang Technological University, 639798, Singapore
`zuocong10@gmail.com, hxwang@ntu.edu.sg`
[2] Faculty of Information Technology, Monash University, Clayton, 3800, Australia
`{shangqi.lai,xingliang.yuan,joseph.liu}@monash.edu`
[3] School of Computer and Information Engineering, Zhejiang Gongshang University, Hangzhou, 310018, Zhejiang, China `chn.junshao@gmail.com`
*Corresponding author

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 forward and backward private DSSE schemes support single keyword queries only, which impedes its prevalence in practice. Until recently, Patranabis et al. (NDSS 2021) introduced a forward and backward private DSSE for conjunctive queries (named `ODXT`) based on the Oblivious Cross-Tags (`OXT`) framework. Unfortunately, its security is not comprehensive for conjunctive queries, and it deploys "lazy deletion", which incurs more communication cost. Besides, it cannot delete a file in certain circumstances. To address these problems, we introduce two forward and backward private DSSE schemes with conjunctive queries (named `SDSSE-CQ` and `SDSSE-CQ-S`). To analysis their security, we present two new levels of backward privacy (named Type-O and Type-O$^-$, where Type-O$^-$ is more secure than Type-O), which describe the leakages of conjunctive queries with `OXT` framework more accurately. Finally, the security and experimental evaluation demonstrate that our proposed schemes achieve better security with comparable computation and communication increase in comparison with `ODXT`.

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

1 Introduction

Dynamic searchable symmetric encryption (DSSE) enables the update of the encrypted database while maintaining searchability, which is a useful tool for protecting user's data that stored on the cloud. However, it leaks more information during the update operations, which can be abused by the attackers [5, 36, 2]. To mitigate the attacks, DSSE schemes are required to maintain two new security notions, namely forward and backward privacy, which are informally introduced by Stefanov et al. [27]. The formal definitions of forward and backward privacy are given by Bost [3] and Bost et al. [4], respectively. In particular, Bost et al. [4] gave three different levels of backward privacy definitions (namely Type-III to Type-I, Type-III is the least secure and Type-I is the most secure) for single keyword queries. Informally, forward privacy requires that the server cannot match newly updated files to previously issued search queries. Correspondingly, during two same search queries, backward privacy does not allow the server to learn the files that were previously added and later deleted. Later, many DSSE schemes with forward/backward privacy have been introduced [22, 31, 15, 9, 1, 37, 30].

Nevertheless, most existing forward and backward private DSSE schemes support single keyword queries only. To make forward and backward private DSSE support conjunctive queries, we need to consider not only the leakages of each keywords but also the leakages of the conjunction of the keywords, which is nontrivial. Very recently, Patranabis et al. [25] proposed a forward and backward private DSSE with conjunctive queries (named `ODXT`) by deploying the framework of `OXT` [7]. Specifically, `OXT` framework has two encrypted datasets (named "TSet" and "XSet"), where "TSet" is used to get files matching the least frequent keyword in a conjunctive query, and "XSet" is used to test if the matching files contain the remaining keywords in the conjunctive queries. For example, given a conjunctive search query $(w_1, w_2, \cdots, w_n)$, the server first searches the "TSet" for keyword w_1 (assume w_1 is the least frequent keyword) and gets the matching results. Then it tests if the results contain the remaining keywords by searching the "XSet".

Table 1. Comparison of results

Scheme	Client Storage	Communication		Computation		Non-interactive	Forward	Backward	Query
		Search	Update	Search	Update	Deletion	Privacy	Privacy	Type
`Aura` [30]	$O(\lvert\mathbf{W}\rvert d)$	$O(n_w - d_w)$	$O(1)$	$O(n_w)$	$O(1)$	✓	✓	Type-II	Single
`ODXT` [25]	$O(\lvert\mathbf{W}\rvert \log D)$	$O(n_q + d_q)$	$O(1)$	$O(n_q + d_q)$	$O(1)$	✗	✓†	Type-II‡	Conjunctive
`SDSSE-CQ`	$O(\lvert\mathbf{W}\rvert d)$	$O(n_q - d_q)$	$O(1)$	$O(n_q)$	$O(1)$	✓	✓	Type-O	Conjunctive
`SDSSE-CQ-S`	$O(\lvert\mathbf{W}\rvert d)$	$O(n_q - d_q)$	$O(1)$	$O(n_q)$	$O(1)$	✓	✓	Type-O$^-$	Conjunctive

$|\mathbf{W}|$ is the 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). Single and Conjunctive stand for single keyword queries and conjunctive queries, respectively. † The forward privacy of `ODXT` is not comprehensive for conjunctive queries. ‡ `ODXT` cannot delete a file in certain circumstance, and 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.

Unfortunately, `ODXT` [25] does not consider the forward privacy for conjunctive queries in certain circumstances. In other words, their security is not comprehensive for conjunctive queries with `OXT` framework. In particular, the authors protect the forward privacy of "TSet", while the "XSet" is not forward private. If a client adds a new keyword/identifier pair for a keyword in $\{w_2, \cdots, w_n\}$ (not the least frequent keyword w_1), the server can test if the newly added file matches the previously issued query, since the server can use the previously issued *xtokens* to test the "XSet". For example, given following queries: $\{1, add, (w_1, ind_1)\}$, $\{2, search, (w_1, w_2)\}$, $\{3, add, (w_2, ind_1)\}$. For the search query happened at time 2 (assume w_1 is the least frequent keyword), there is no matching files. However, according to ODXT, after the update query at time 3, the server can learn that ind_1 contains the keywords w_1 and w_2 by using the previously issued query $\{2, search, (w_1, w_2)\}$, which is contradict to the requirement of forward privacy.

In addition, ODXT deploys the "lazy deletion" technique from [4], where the deletion is achieved by adding an indicator "del" to the add operation. Then the server returns all the search results to the client, and the client filters out the deleted files, which requires more interactions between the client and the server. Besides, ODXT cannot delete a file in certain circumstance, because the deletion tag is not added for the files contain the least frequent keyword.

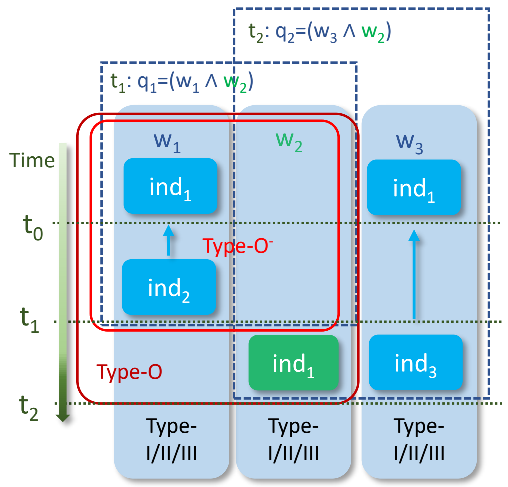

Fig. 1. Illustration of Type-O and Type-O$^-$

Our Contributions. To address the problems, in this paper, we introduce two schemes (named SDSSE-CQ and SDSSE-CQ-S). Table 1 compares our results with the related works. The concrete contributions are as follows:

– We first give a new DSSE with conjunctive queries (named SDSSE-CQ) based on the framework of OXT [7]. In particular, to reduce the interactions between the client and the server, we apply the state-of-the-art forward and backward private DSSE (termed Aura) with non-interactive deletion from [30][4]. In addition, we guarantee the forward privacy of both "TSet" and

[4] Note that Aura only supports single keyword queries, and it can be replaced with any forward and backward private DSSE for single keyword queries.

"XSet". To further reduce the leakages of `SDSSE-CQ`, we introduce another scheme (named `SDSSE-CQ-S`) by increasing the security of the "XSet". See Section 4 for details.
- To precisely quantify the leakages of our proposed schemes, we introduce two new levels of backward privacy definitions for conjunctive queries with `OXT` framework (named Type-O and Type-O$^-$, where Type-O$^-$ is more secure than Type-O). Informally, as illustrated Fig. 1, the key difference between Type-O and Type-O$^-$ is that if newly searched "xtag"s (e.g., w_2 from $q_2 = (w_3, w_2)$) can be used by previously issued queries with the same "xterm" (e.g., $q_1 = (w_1 \wedge w_2)$). Each keyword is protected by a forward and backward private DSSE 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[5]. Note that `ODXT` does not protect the forward privacy of the "xtag"s (e.g., (w_2, ind_1) can be searched by the previously issued query q_1 without the help of q_2).
- Finally, the security and experimental analysis demonstrate the security and practicality of our proposed schemes. Compared with `ODXT`, our proposed schemes achieve better security with little computation and communication increase. See Section 6 for details.

1.1 Related Work

Song et al. [26] first deployed symmetric key encryption to address keyword search over encrypted data, which is known as searchable symmetric encryption (SSE). However, its search time is linear with the number of keyword/identifier pairs. To improve the search efficiency, Goh [18] proposed a scheme with secure indexes, where the search time is linear with the number of files. To further improve the search efficiency, Curtmola et al. [10] gave a sublinear search time SSE by deploying an inverted index data structure. In addition, they also formalized the SSE security model (i.e., Real v.s Ideal), which has been adopted by the following works. Later, many SSE schemes with different improvements have been introduced (e.g., rich queries [7, 16, 23, 14, 39], dynamism [21, 6], multi-client model [10, 29], locality [8, 24, 13], small client storage [12], etc.).

To make SSE support update, dynamic SSE (DSSE) [21, 6] has been introduced. However, these schemes leak extra information during updates, and the information can be abused by adversaries to compromise data privacy [5, 36, 19, 20, 2], which highlights the importance of forward and backward privacy. They are informally introduced by Stefanov et al. [27]. Bost [3] has formally defined forward privacy, and the formal backward privacy is defined by Bost et al. [4]. In particular, they defined Type-I, Type-II, and Type-III backward privacy, where Type-I is more secure than Type-II and Type-II is more secure than Type-III. In addition, they gave several DSSE schemes with different levels of backward privacy. Specifically, the Type-I backward private (called `MONETA`)

[5] It depends on which level of backward privacy the underlying DSSE achieves. For example, if we deploy `Aura`, then it becomes Type-II backward privacy.

shows the feasibility of the Type-I backward private DSSE, which is based on the TWORAM [17]. Furthermore, They proposed `FIDES` with Type-II backward privacy. $\texttt{DIANA}_{del}$ and `Janus`, achieve Type-III backward privacy, are more efficient. To further improve the efficiency of `Janus`, Sun et al. [31] introduced `Janus++` with Type-III backward privacy by replacing the (public-key) puncturable encryption (PE) of `Janus` with their proposed symmetric PE (SPE). Concurrently, Chamani et al. [9] introduced a forward and Type-II backward private DSSE `MITRA`, while it needs to generate search tokens for each entry. To reduce the search tokens, they also deployed the Path ORAM [28] to give new constructions for forward and backward private DSSE (namely `ORION` and `HORUS`). To achieve a stronger level of backward privacy efficiently, Zuo et al. [37] introduced `FB-DSSE` by using the simple symmetric encryption with homomorphic addition and bitmap index. In particular, their scheme achieves Type-I^- backward privacy, which is somewhat stronger than Type-I[6]. Very recently, Sun et al. [30] introduced a forward and backward private DSSE (named `Aura`), which achieves non-interactive deletion.

However, the aforementioned forward and backward private DSSE schemes support single keyword queries only. To address the problem, Zuo et al. [39] 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. Later, Wang et al. [32] proposed a generic forward private DSSE with range queries based on the `SchemeA`. In addition, they extended the scheme to achieve backward privacy by using the "lazy deletion" technique from [4], which is less efficient. To give a more efficient forward and backward private DSSE with range query, Zuo et al. [38] introduced `FBDSSE-RQ`, which applies the framework of `FB-DSSE` [37]. Very recently, Patranabis et al. [25] introduced a forward and backward DSSE for conjunctive queries (`ODXT`), which is based on the framework of `OXT` [7]. However, as mentioned before, its security is not comprehensive for conjunctive queries, and it cannot delete a file in certain circumstance. It is not an easy task to make the DSSE with `OXT` framework support full forward and backward privacy, because we need to quantify both the leakages of each keyword in a conjunctive query and the keyword conjunction leakages, which is complicate.

1.2 Organization

The remaining sections of this paper are organized as follows. In Section 2, we give the necessary background and preliminaries. In Section 3, we define the DSSE, its security model and introduce `Aura`. In Section 4, we give our forward and backward DSSE schemes with conjunctive queries. The security analysis is given in Section 5. Section 6 gives the experimental evaluation of our schemes as well as the `ODXT`. Finally, Section 7 concludes the work.

[6] Type-I^- does not leak the insertion time of the matching files, while Type-I does.

2 Preliminaries

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

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 $\mathbf{Adv}_{\mathcal{A}}^{\mathtt{DDH}}(\lambda)$ 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,

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

2.2 Symmetric Encryption

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

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

Correctness. An SE scheme is *perfectly correct* if for all message $m \in \mathcal{M}$, secret key $k \in \mathcal{K}$, and $ct \leftarrow \mathtt{SE{\cdot}Enc}(k, m)$, it holds that $\Pr[\mathtt{SE{\cdot}Dec}(k, ct)] = 1$.

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

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

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 adaptively issue a polynomial number of encryption queries. For each $m \in \mathcal{M}$, the challenger returns $ct \leftarrow \mathtt{SE{\cdot}Enc}(k, m)$.

3 DSSE Definition and Security Model

A database $\mathtt{DB}$ stores a list of file-identifier/keyword-set pairs or $\mathtt{DB}= (ind_i, \mathbf{W}_i)_{i=1}^{D}$, where $ind_i \in \{0, 1\}^{\lambda}$ is the file identifier, $\mathbf{W}_i$ is the keyword set of file f_i and D is the total number of files in $\mathtt{DB}$. We denote the collection of all distinct keywords in $\mathtt{DB}$ by $\mathbf{W} = \cup_{i=1}^{D} \mathbf{W}_i$. The notation $|\mathbf{W}|$ stands for the total number of keywords in the set $\mathbf{W}$ (or cardinality of the set). The total number of file-identifier/keyword pairs is denoted by $N = \sum_{i=1}^{D} |\mathbf{W}_i|$.

A set of files that satisfy a conjunctive query q is denoted by $\mathtt{DB}(q)$. For a search query q, the result is a set of file identifiers represented in $\mathtt{DB}(q)$. For an update query u, a file index ind corresponding to a keyword w is updated [7].

3.1 DSSE Definition

A DSSE scheme consists of an algorithm **Setup**, two protocols **Search** and **Update** that are executed between a client and a server. They are described as follows:

- $(\mathtt{EDB}, \sigma) \leftarrow$ **Setup**$(1^\lambda, \mathtt{DB})$: For a security parameter λ and a database $\mathtt{DB}$, the algorithm outputs a pair: an encrypted database $\mathtt{EDB}$ and a state σ. $\mathtt{EDB}$ is stored by the server and σ is kept by the client.
- $(\mathcal{I}, \perp) \leftarrow$ **Search**$(q, \sigma; \mathtt{EDB})$: For a state σ, the client issues a search query q and interacts with the server who holds $\mathtt{EDB}$. At the end of the protocol, the client outputs a set of file identifiers $\mathcal{I}$ that match q and the server outputs nothing.
- $(\sigma', \mathtt{EDB}') \leftarrow$ **Update**$(\sigma, op, in; \mathtt{EDB})$: For a state σ, the operation $op \in \{add, del\}$ and a collection of $in = (ind, \mathbf{w})$ pairs[8], the client requests the server (who holds $\mathtt{EDB}$) to update database by adding/deleting files specified by the collection in. Finally, the protocol returns an updated state σ' to the client and an updated encrypted database $\mathtt{EDB}'$ to the server.

Remark. There are two result models for (D)SSE schemes in the literature. In the first one (considered in the work [7]), the server returns encrypted file identifiers, so the client needs to decrypt them. In the second one (studied in the work [3]), the server returns the file identifiers to the client as plaintext. In this work, we consider the first variant, where the protocol returns encrypted file identifiers.

3.2 Security Model

DSSE security is modeled by the interaction between the Real and Ideal worlds called $\mathtt{REAL}$ and $\mathtt{IDEAL}$, respectively. The behavior of $\mathtt{REAL}$ is exactly the same as the original DSSE. However, $\mathtt{IDEAL}$ reflects a behavior of a simulator $\mathcal{S}$, which takes the leakages of the original DSSE as input. The leakages are defined by the function $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Search}, \mathcal{L}^{Update})$, which details what information the adversary $\mathcal{A}$ can learn during execution of the **Setup** algorithm, **Search** and **Update** protocols.

[7] If you want to update more file-identifier/keyword-set pairs, you can update many times.

[8] Note that in this paper, as mentioned before, for every update, we update one file-identifier/keyword pair.

If the adversary $\mathcal{A}$ can distinguish $\mathtt{REAL}$ from $\mathtt{IDEAL}$ with a negligible advantage, we can say that leakage of information is restricted to the leakage $\mathcal{L}$. More formally, we consider the following security game. The adversary $\mathcal{A}$ interacts with one of the two worlds $\mathtt{REAL}$ or $\mathtt{IDEAL}$ which are described as follows:

– $\mathtt{REAL}_{\mathcal{A}}(\lambda)$: On input a database $\mathtt{DB}$, which is chosen by the adversary $\mathcal{A}$, it outputs $\mathtt{EDB}$ to the $\mathcal{A}$ by running $\mathbf{Setup}(\lambda, \mathtt{DB})$. $\mathcal{A}$ performs search queries q (or update queries (op, in)). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0,1\}$.
– $\mathtt{IDEAL}_{\mathcal{A},\mathcal{S}}(\lambda)$: Simulator $\mathcal{S}$ outputs the simulated $\mathtt{EDB}$ with input $\mathcal{L}^{Setup}(\lambda, \mathtt{DB}))$. For search queries q (or update queries (op, in)) generated by the adversary $\mathcal{A}$, the simulator $\mathcal{S}$ replies by using the leakage function $\mathcal{L}^{Search}(q)$ (or $\mathcal{L}^{Update}(op, in)$). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0,1\}$.

Definition 1. *Given a DSSE scheme and the security game described above. The scheme 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,*

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

3.3 DSSE for Single Keyword Queries

Our schemes can be constructed from any forward and backward private DSSE for single keyword queries. As mentioned before, `ODXT` deploys "lazy deletion" to achieve backward privacy, which incurs more interactions and is not efficient. To mitigate this, we deploy the state-of-the-art DSSE scheme for single keyword queries from [30] (named `Aura`), which achieves non-interactive deletion and is forward and Type-II backward private. Specifically, `Aura` consists of the following polynomial-time algorithm and protocols $\Sigma = (\Sigma\text{·}\mathbf{Setup}, \Sigma\text{·}\mathbf{Search}, \Sigma\text{·}\mathbf{Update})$:

– $(\sigma, \mathtt{EDB}) \leftarrow \Sigma\text{·}\mathbf{Setup}(1^{\lambda})$: The algorithm is run by the client. On input the security parameter λ, it outputs the secret state σ and the encrypted database $\mathtt{EDB}$.
– $(\sigma', \mathtt{EDB}') \leftarrow \Sigma\text{·}\mathbf{Update}(op, (w, ind), \sigma; \mathtt{EDB})$: The protocol runs between a client and a server. The client inputs an operation op, a keyword/identifier pair (w, ind), the secret state σ and the server inputs the encrypted database $\mathtt{EDB}$. Finally, the client outputs an updated state σ' and the server outputs an updated encrypted database $\mathtt{EDB}'$.
– $\mathbf{Res} \leftarrow \Sigma\text{·}\mathbf{Search}(w, \sigma; \mathtt{EDB})$: The protocol runs between a client and a server. The client inputs a search keyword w, the state σ and the server inputs the encrypted database $\mathtt{EDB}$. Finally, the client outputs the search result $\mathbf{Res}$ and the server outputs nothing.

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

Security. According to [30], `Aura` achieves forward and Type-II backward privacy. According to definition 3, we say `Aura` is forward and Type-II backward private if for every probabilistic polynomial time (PPT) adversary $\mathcal{A}$, its advantage

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

is negligible. We refer readers to [30] for details.

4 Our Constructions

In this section, we give 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.

4.1 Forward and Backward Private DSSE for Conjunctive Queries

As mentioned before, the security model of `ODXT`[25] is not comprehensive for conjunctive queries with `OXT`[7] 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 between the client and the server, we deploy the general construction (named `Aura`) from [30], which supports non-interactive deletion. Note that `Aura` only supports single keyword queries. In addition, we can replace `Aura` with any forward and backward private DSSE for single keyword queries.

Let $\Sigma = (\Sigma\cdot\mathbf{Setup}, \Sigma\cdot\mathbf{Search}, \Sigma\cdot\mathbf{Update})$ be the forward and backward private DSSE scheme from [30]. We introduce our forward and backward private DSSE with conjunctive queries $\texttt{SDSSE-CQ} = (\texttt{SDSSE-CQ}\cdot\mathbf{Setup}, \texttt{SDSSE-CQ}\cdot\mathbf{Update}, \texttt{SDSSE-CQ}\cdot\mathbf{Search})$, which is defined in algorithm 1. For simplicity, we assume w_1 is the least frequent keyword. The algorithms are described as follows:

- $(\texttt{EDB}, \sigma) \leftarrow \texttt{SDSSE-CQ}\cdot\mathbf{Setup}(1^{\lambda})$: The algorithm is run by a client. He/she takes the security parameter λ as input. Then he/she chooses a secret key k for keyed PRF F and key k_x, k_i, k_z for keyed PRF F_p (with range in Z_p^*). Moreover, he/she sets an empty map $\mathbf{CT}$, which stores keyword/counter (w/c) pairs. In addition, he/she sets two `Aura` instances, which are used for for "TSet" and "XSet", respectively. Finally, he/she outputs encrypted database $\texttt{EDB} = (\texttt{EDB}_T, \texttt{EDB}_X)$ and the state $\sigma = (k, k_x, k_i, k_z, \mathbf{CT}, \sigma_T, \sigma_X)$, and the client keeps the state σ secret.
- $(\sigma', \texttt{EDB}') \leftarrow \texttt{SDSSE-CQ}\cdot\mathbf{Update}(op, w, ind, \sigma; \texttt{EDB})$: The protocol runs between a client and a server. The 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

Algorithm 1 `SDSSE-CQ`

```
SDSSE-CQ·Setup($1^\lambda$)
1: $k \xleftarrow{\$} \{0,1\}^\lambda$ for PRF $F$, $k_x, k_i, k_z \xleftarrow{\$} \{0,1\}^\lambda$ for PRF $F_p$ (with range in $Z_p^*$)
2: $\mathbf{CT} \leftarrow$ empty map
3: $(\sigma_T, \mathtt{EDB}_T) \leftarrow \Sigma\cdot\mathbf{Setup}(1^\lambda)$
4: $(\sigma_X, \mathtt{EDB}_X) \leftarrow \Sigma\cdot\mathbf{Setup}(1^\lambda)$
5: return $(\mathtt{EDB} = (\mathtt{EDB}_T, \mathtt{EDB}_X)$, $\sigma = (k, k_x, k_i, k_z, \mathbf{CT}, \sigma_T, \sigma_X))$

SDSSE-CQ·Update($op, w, ind, \sigma$; EDB)
Client:
1: $c \leftarrow \mathbf{CT}[w]$
2: if $c = \perp$ then
3:     $c \leftarrow -1$
4: end if
5: $c \leftarrow c + 1$, $\mathbf{CT}[w] \leftarrow c$
6: $k_w \leftarrow F(k, w)$, $e \leftarrow \mathtt{SE}\cdot\mathtt{Enc}(k_w, ind)$
7: $xind \leftarrow F_p(k_i, ind)$, $z \leftarrow F_p(k_z, w||c)$, $y \leftarrow xind \cdot z^{-1}$
8: $xtag \leftarrow g^{F_p(k_x, w)\cdot xind}$
Client ↔ Server:
9: Run$\Sigma\cdot\mathbf{Update}(op, (w, e||y||c), \sigma_T; \mathtt{EDB}_T)$
10: Run$\Sigma\cdot\mathbf{Update}(op, (w, xtag), \sigma_X; \mathtt{EDB}_X)$

SDSSE-CQ·Search($q = (w_1, \cdots, w_n), \sigma$; EDB)
Client:
1: $c \leftarrow \mathbf{CT}[w]$, $k_{w_1} \leftarrow F(k, w_1)$
2: if $c = \perp$ then
3:     return $\varnothing$
4: end if
5: for $i = 0$ to $c$, $j = 2$ to $n$ do
6:     $xtoken[i, j] \leftarrow g^{F_p(k_z, w_1||i)\cdot F_p(k_x, w_j)}$
7: end for
8: Send $xtoken$ to the server.
Client ↔ Server:
9: $\mathbf{Res}_T \leftarrow \Sigma\cdot\mathbf{Search}(w_1, \sigma_T; \mathtt{EDB}_T)$
10: if $\mathbf{Res}_T = \perp$ then
11:     return $\varnothing$
12: end if
13: $\mathbf{XSet} \leftarrow$ empty set
14: for $j = 2$ to $n$ do
15:     $\mathbf{Res}_X \leftarrow \Sigma\cdot\mathbf{Search}(w_j, \sigma_X; \mathtt{EDB}_X)$
16:     if $\mathbf{Res}_X = \perp$ then
17:         return $\varnothing$
18:     end if
19:     $\mathbf{XSet} \leftarrow \mathbf{XSet} \cup \mathbf{Res}_X$
20: end for
Server:
21: $\mathbf{Res} \leftarrow$ empty set
22: for each $(e||y||c) \in \mathbf{Res}_T$ do
23:     $flag \leftarrow true$
24:     for $j = 2$ to $n$ do
25:         if $xtoken[c, j]^y \notin \mathbf{XSet}$ then
26:             $flag \leftarrow false$
27:         end if
28:     end for
29:     if $flag$ then
30:         $\mathbf{Res} \leftarrow \mathbf{Res} \cup e$
31:     end if
32: end for
33: Send $\mathbf{Res}$ to the client.
Client:
34: for each $e \in \mathbf{Res}$ do
35:     $ind \leftarrow \mathtt{SE}\cdot\mathtt{Dec}(k_{w_1}, e)$
36: end for
```

xtag. Then the client interacts with the server to update $(e||y||c)$ and $xtag$ corresponding to keyword w by using the $\Sigma\cdot$**Update**[9]. Finally, the client outputs an updated state σ' and the server outputs an updated encrypted database $\mathtt{EDB}'$.

– $\mathcal{I} \leftarrow$ `SDSSE-CQ`·**Search**$(q = (w_1, w_2, \cdots, w_n), \sigma;\ \mathtt{EDB})$: The protocol runs between a client and a server. The client inputs a conjunctive query $q = (w_1, w_2, \cdots, w_n)$ and a state σ, and the server inputs EDB. Firstly, the client gets c from **CT** corresponding to keyword w_1 (we assume w_1 is the least

[9] Note that the update time of `Aura` is leaked. In other words, the counter c is implicitly leaked in `Aura`. Hence, the counter c does not incur a new leakage.

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. After that, the client interacts with the server to search w_1 from $\texttt{EDB}_T$ and $(w_2, \cdots, w_n)$ from $\texttt{EDB}_X$ through Σ·**Search**. The server retrieves all the file identifiers corresponding to w_1 and tests if them 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.

Remark. `SDSSE-CQ` achieves forward privacy for DSSE with conjunctive queries because it protects the forward privacy of both "TSet" and "XSet", where `ODXT` does not. Nevertheless, the newly searched *xtag*s still can be used by previously issued search queries (with different least frequent keywords), and we call this Type-O backward privacy. In particular, given the following queries: $\{1, add, (w_1, ind_1)\}$, $\{2, search, (w_1, w_2)\}$, $\{3, add, (w_2, ind_1)\}$, $\{4, search, (w_3, w_2)\}$ (assume w_1, w_3 are the least frequent keywords). For the search query happened at time 2, there is no matching files. After the search query happened at time 4, the server can learn that ind_1 contains the keywords w_1 and w_2 by using the previously issued query $\{2, search, (w_1, w_2)\}$. This is due to the fact that the newly generated *xtag*s are revealed to the server and can be used by previously issued search queries.

4.2 Stronger Backward Privacy

As mentioned before, `SDSSE-CQ` achieves Type-O backward privacy. To further reduce the leakage, we introduce a stronger level backward privacy (named Type-O$^-$), where the newly generated *xtag*s cannot be used by previously issued search queries. To achieve the stronger level of backward privacy, we add a new random number (corresponding to the keyword/counter pair) to the *xtag*. Specifically, we introduce `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 red.

- $(\texttt{EDB}, \sigma) \leftarrow$ `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.
- $(\sigma', \texttt{EDB}') \leftarrow$ `SDSSE-CQ-S`·**Update**$(op, w, ind, \sigma; \texttt{EDB})$: This protocol is almost the same as the `SDSSE-CQ`·**Update** except that it puts a new randomness (corresponding to the update keyword/counter $w||c$, $F_p(k'_x, w||c)^{-1}$) to the *wxtag* ($\leftarrow g^{F_p(k_x,w)\cdot xind\cdot F_p(k'_x,w||c)^{-1}}$), which is used to avoid the newly generated *wxtag*s to be used by previously issued search queries (with different least frequent keyword).
- $\mathcal{I} \leftarrow$ `SDSSE-CQ-S`·**Search**$(q = (w_1, w_2, \cdots, w_n), \sigma;\ \texttt{EDB})$: This protocol is similar to the `SDSSE-CQ`·**Search**. 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* ($wxtoken[i,j] \leftarrow g^{F_p(k_z,w_1||i)\cdot F_p(k_x,w_j)\cdot F_p(k'_z,w_1)}$) and generates new tokens $wxk[j][k] = F_p(k'_x, w_j||c_j) \cdot F_p(k'_z, w_1)$. When the server try

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

```
SDSSE-CQ-S·Setup($1^\lambda$)
1: $k \xleftarrow{\$} \{0,1\}^\lambda$ for PRF $F$, $k_x, k_i, k_z, k'_x, k'_z \xleftarrow{\$} \{0,1\}^\lambda$ for PRF $F_p$ (with range in $Z_p^*$)
2: $\mathbf{CT} \leftarrow$ empty map
3: $(\sigma_T, \mathtt{EDB}_T) \leftarrow \Sigma$·Setup($1^\lambda$)
4: $(\sigma_X, \mathtt{EDB}_X) \leftarrow \Sigma$·Setup($1^\lambda$)
5: return $(\mathtt{EDB} = (\mathtt{EDB}_T, \mathtt{EDB}_X), \sigma = (k, k_x, k_i, k_z, k'_x, k'_z, \mathbf{CT}, \sigma_T, \sigma_X))$

SDSSE-CQ-S·Update($op, w, ind, \sigma; \mathtt{EDB}$)
Client:
1: $c \leftarrow \mathbf{CT}[w]$
2: if $c = \perp$ then
3:     $c \leftarrow -1$
4: end if
5: $c \leftarrow c + 1$, $\mathbf{CT}_T[w] \leftarrow c$
6: $k_w \leftarrow F(k, w)$, $e \leftarrow \mathtt{SE{\cdot}Enc}(k_w, ind)$
7: $xind \leftarrow F_p(k_i, ind)$, $z \leftarrow F_p(k_z, w||c)$, $y \leftarrow xind \cdot z^{-1}$
8: $wxtag \leftarrow g^{F_p(k_x,w) \cdot xind \cdot F_p(k'_x, w||c)^{-1}}$
Client ↔ Server:
9: Run $\Sigma$·Update($op, (w, e||y||c), \sigma_T; \mathtt{EDB}_T$)
10: Run $\Sigma$·Update($op, (w, wxtag||c), \sigma_X; \mathtt{EDB}_X$)

SDSSE-CQ·Search($q = (w_1, \cdots, w_n), \sigma; \mathtt{EDB}$)
Client:
1: $c \leftarrow \mathbf{CT}[w]$, $k_{w_1} \leftarrow F(k, w_1)$
2: if $c = \perp$ then
3:     return $\varnothing$
4: end if
5: for $i = 0$ to $c$, $j = 2$ to $n$ do
6:     $wxtoken[i, j] \leftarrow g^{F_p(k_z, w_1||i) \cdot F_p(k_x, w_j) \cdot F_p(k'_z, w_1)}$
7: end for
8: for $j = 2$ to $n$ do
9:     $c_j \leftarrow \mathbf{CT}[w_j]$
10:    if $c_j = \perp$ then
11:        return $\varnothing$
12:    end if
13:    for $k = 0$ to $c_j$ do
14:        $wxt[j][k] \leftarrow F_p(k'_x, w_j||k) \cdot F_p(k'_z, w_1)$
15:    end for
16: end for
17: Send $(wxtoken, wxt)$ to the server.
Client ↔ Server:
18: $\mathbf{Res}_T \leftarrow \Sigma$·Search($w_1, \sigma_T; \mathtt{EDB}_T$)
19: if $\mathbf{Res}_T = \perp$ then
20:     return $\varnothing$
21: end if
22: $\mathbf{WXSet} \leftarrow$ empty set
23: for $j = 2$ to $n$ do
24:     $\mathbf{Res}_X \leftarrow \Sigma$·Search($w_j, \sigma_X; \mathtt{EDB}_X$)
25:     if $\mathbf{Res}_X = \perp$ then
26:         return $\varnothing$
27:     end if
28:     for each $wxtag||c_j \in \mathbf{Res}_X$ do
29:         $\mathbf{WXSet} \leftarrow \mathbf{WXSet} \cup wxtag^{wxt[j][c_j]}$
30:     end for
31: end for
Server:
32: $\mathbf{Res} \leftarrow$ empty set
33: for each $(e||y||c) \in \mathbf{Res}_T$ do
34:     $flag \leftarrow true$
35:     for $j = 2$ to $n$ do
36:         if $wxtoken[c, j]^y \notin \mathbf{WXSet}$ then
37:             $flag \leftarrow false$
38:         end if
39:     end for
40:     if $flag$ then
41:         $\mathbf{Res} \leftarrow \mathbf{Res} \cup e$
42:     end if
43: end for
44: Send $\mathbf{Res}$ to the client.
Client:
45: for each $e \in \mathbf{Res}$ do
46:     $ind \leftarrow \mathtt{SE{\cdot}Dec}(k_{w_1}, e)$
47: end for
```

to recover the new "$xtag$" from $wxtag$ ($wxtag^{wxt}$). Then 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*". 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.

5 Security Analysis

In this section, we first define the common leakage functions in DSSE. Then we give the forward and backward privacy for our conjunctive queries. Finally, we give the security analysis of our proposed schemes.

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 list of queries Q, we define a search pattern

$$\mathtt{sp}(q) = \{t : \{\mathtt{sp}(w)\}_{w \in q}\}_{q \in Q},$$

where t is a timestamp and $\mathtt{sp}(w) = (t : (t, w))$ leaks the timestamp of a keyword w. The search pattern leaks the repetition of search queries on each keyword $w \in q$. We also define a result pattern

$$\mathtt{rp}(q) = \{\mathcal{I}\}_{q \in Q},$$

where $\mathcal{I}$ represents all file identifiers that match the conjunctive query (i.e., $\mathtt{DB}(q)$).

5.1 Forward Privacy

Informally, for any adversary who may continuously observe the interactions between the server and the client, forward privacy guarantees that an update does not leak information about the newly added files that match the previously issued queries. In 2016, Bost [3] gave a formal forward privacy definition, which is described below.

Definition 2. *A $\mathcal{L}$-adaptively-secure DSSE scheme is forward private, if the update leakage function $\mathcal{L}^{Update}$ can be written as*

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

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

In [25], Patranabis et al. deployed the OXT [7] framework to achieve forward and backward private DSSE 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.

5.2 Backward Privacy

Informally, during two same search queries q, backward privacy ensures that the server cannot learn the files previously added and later deleted. Note that the two search queries should be the same. Otherwise, the server cannot tell whether the missing files do not match the search query or are deleted. In [4], Bost et al. gave three different levels of backward privacy (namely, Type-I to Type-III, from most secure to least secure) for single keyword queries, which can not describe the leakages of conjunctive queries properly. For conjunctive queries with `OXT` framework, two search queries may have same "xterm" (i.e., $q_1 = (w_1, w_2)$ and $q_2 = (w_3, w_2)$, where w_2 is the "xterm". If there is an update query between q_1 and q_2 for keyword w_2. After q_2, the server can use the previous query q_1 to test if the new added file contains keyword w_1. `SDSSE-CQ` contains such leakage. To avoid this leakage, we introduce `SDSSE-CQ-S`. To formally describe this information, we define two new backward privacy definitions named Type-O and Type-O$^-$, where Type-O$^-$ is more secure. Note that 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, for each keyword in Type-O/O$^-$, we mainly focus on the Type-II backward privacy. Specifically,

- Type-O$^-$: Given two same consecutive conjunctive queries q, it leaks file identifiers that currently match a query q. It also leaks the number of matching files of each $w_1 \wedge W_{w_1}$ pair, where $q = (w_1, w_2, \cdots, w_n)$[10], W_{w_1} is the conjunction of any subset of $\{w_2, \cdots, w_n\}$ corresponding to the current least frequent keyword w_1. For each keyword $w \in q$, it leaks the total number of updates and the corresponding update times[11].
- Type-O: Apart from the leakages in Type-O$^-$, it leaks the number of matching files of each $w_1 \wedge W$ pair, where W is the conjunction of any subset of $\{w_2, \cdots, w_n\}$.

[10] We assume w_1 is the least frequent keyword.

[11] This is dependent on the Type-II backward privacy. For other types of backward privacy (e.g., Type-III), it can be changed accordingly.

To formally define the notion, we need to introduce some new leakage functions. For a conjunctive query q, $\texttt{Time}(q)$ shows update time t for each keyword $w \in q$. Formally,

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

Let $\texttt{size}(w_1 \wedge W) = \{|\texttt{DB}(w_1 \wedge W)|\}$ denotes the number of matching files for any conjunctive queries with the form $w_1 \wedge W$, where $q = (w_1, w_2, \cdots, w_n)$, W is the conjunction of any subset of $\{w_2, \cdots, w_n\}$, and $|\texttt{DB}(w_1 \wedge W)|$ denotes the number of matching files for all possible combinations (e.g., $(w_1 \wedge w_2)$, $(w_1 \wedge w_2 \wedge w_3)$, etc)[12]. Similarly, we can define $\texttt{size}(w_1 \wedge W_{w_1}) = \{|\texttt{DB}(w_1 \wedge W_{w_1})|\}$, where W_{w_1} denotes the keywords in W correspond to the current least frequent keyword w_1. For example, we initially assume file ind_1 has keywords w_1 and w_3. Now, we have a series of queries $(1, search, q = (w_1, w_2))$, $(2, add, (w_2, ind_1))$, $(3, search, q = (w_3, w_2))$. After the search time 3, we have $\texttt{size}(w_1 \wedge W) = 1$ and $\texttt{size}(w_1 \wedge W_{w_1}) = 0$, where W is $\{w_2\}$.

Definition 3. *A $\mathcal{L}$-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*
$\mathcal{L}^{Search}(q) = \mathcal{L}''(\texttt{sp}(q), \texttt{rp}(q), \texttt{Time}(q), \texttt{size}(w_1 \wedge W_{w_1}))$,

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

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

5.3 Proofs

In this subsection, we give the security proofs of our proposed schemes. Note that SDSSE-CQ and SDSSE-CQ-S do not leak the final result rp. Hence the following theorems do not contain this leakage. We first give the proof of SDSSE-CQ, which achieves forward and Type-O backward privacy. Formally,

Theorem 1. *(Adaptive forward and Type-O backward privacy of SDSSE-CQ). Let Σ be forward and backward private, F, F_p be secure PRFs, DDH assumption holds over $\mathbb{G}$, SE be a IND-CPA secure symmetric encryption. We define $\mathcal{L}_{\texttt{SDSSE-CQ}} = (\mathcal{L}^{Update}_{\texttt{SDSSE-CQ}}, \mathcal{L}^{Search}_{\texttt{SDSSE-CQ}})$, where $\mathcal{L}^{Update}(op, w, ind) = op$ and $\mathcal{L}^{Search}(q) = (\texttt{sp}(q), \texttt{Time}(q), \texttt{size}(w_1 \wedge W))$. Then SDSSE-CQ is $\mathcal{L}_{\texttt{SDSSE-CQ}}$-adaptively forward and Type-O backward private.*

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 $w_1 \wedge W$ (W denotes the conjunction of any subset of $\{w_2, \cdots, w_n\}$). Hence, SDSSE achieves Type-O backward privacy.

[12] The number of matching files for each keyword w can be deduced from the $\texttt{Time}(q)$.

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

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

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

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. For a new keyword w (resp., w, ind, $w||c$), they choose new values and store them in table `Key` (resp., $\mathtt{G}_x$, $\mathtt{G}_i$, $\mathtt{G}_z$). 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

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

Game G_2: In this game, similar to [7], 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 we have

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

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`. Then we have

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

Game G_4: In this game, we can use the leakage of $\mathtt{Time}(q)$ and $\mathtt{size}(w_1 \wedge W)$ to choose and set the random values for *xtoken*s properly. This has no influence to the distribution of G_4, Then we have

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

Game G_5: For the update and search of `Aura` [30], we can use the same technique to simulate the corresponding values by using the leakages defined in $\mathcal{L}_{\mathtt{SDSSE\text{-}CQ}}$. If an adversary $\mathcal{A}$ can distinguish G_5 from G_4, then we can build an adversary $\mathcal{B}_4$ to break the forward and backward privacy of `Aura`. Therefore, we have

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

Simulator: Now, we can use the $\mathtt{sp}(q) = min\{\mathtt{sp}(w)\}_{w \in q}$ to simulate the search queries. Moreover, the construction of encrypted database `EDB` can be properly simulated by using the leakage of $\mathtt{Time}(q)$ and $\mathtt{size}(w_1 \wedge W)$ 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

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

Finally, we can conclude that the advantage of any adversary $\mathcal{A}$ attacking SDSSE-CQ is

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

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

□

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 a stronger level of backward privacy (named 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 of SDSSE-CQ-S. Then new "*xtag*" (termed as *wxtag*) can only be used to test the remaining keywords corresponding to the current least frequent keyword w_1. Formally,

Theorem 2. *(Adaptive forward and Type-O$^-$ backward privacy of SDSSE-CQ-S). Let Σ be forward and backward private, F, F_p be secure PRFs, DDH assumption holds over $\mathbb{G}$, SE be a IND-CPA secure symmetric encryption. We define $\mathcal{L}_{\mathtt{SDSSE\text{-}CQ}}=(\mathcal{L}_{\mathtt{SDSSE\text{-}CQ}}^{Update},\mathcal{L}_{\mathtt{SDSSE\text{-}CQ}}^{Search})$, where $\mathcal{L}^{Update}(op,w,ind)=op$ and $\mathcal{L}^{Search}(q)=(\mathtt{sp}(q),\mathtt{Time}(q),\mathtt{size}(w_1\wedge W_{w_1}))$. Then we have SDSSE-CQ-S is $\mathcal{L}_{\mathtt{SDSSE\text{-}CQ\text{-}S}}$-adaptively forward and Type-O$^-$ backward private.*

Proof. Similar to the proof of Theorem 1, we can set a series of games from $\mathtt{REAL}_{\mathcal{A}}^{\mathtt{SDSSE\text{-}CQ\text{-}S}}(\lambda)$ to $\mathtt{IDEAL}_{\mathcal{A},\mathcal{S}}^{\mathtt{SDSSE\text{-}CQ\text{-}S}}(\lambda)$ 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*. As a result, we can conclude that the advantage of any adversary $\mathcal{A}$ attacking SDSSE-CQ-S is

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

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

□

6 Experimental Analysis

In this section, we give the experimental analysis of our proposed schemes and the state-of-the-art ODXT.

6.1 Implementation and Settings

The proposed schemes and `ODXT` are implemented with JAVA, and we use `Aura` [30] to instantiate our schemes as `Aura` achieves non-interactive deletion. For `Aura`, we use the symmetric one-way technique of `FB-DSSE` from [37] as the underlying forward private DSSE[13]. Similar to [30], the parameters for the bloom filter used in `Aura` are set as follows: false positive (10^{-4}), number of bytes (24729) and number of hash functions (6). 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 [30] for details). We leverage the group $\mathbb{G}_T$[14] of JPBC [11] with the input *a.properties* to enable the elliptical curve-based cryptographic operations (e.g., group multiplication and exponentiation) involved in conjunctive queries.

Our evaluation platform is a Ubuntu 20.04.3 LTS workstation with Intel @Xeon(R) W-2123 CPU 3.60GHz with 8 cores and 32GB RAM. We construct a synthesis dataset with 10^5 files, and each file contains at least two same keywords. In the following evaluations, we aim to demonstrate the practicality of the proposed schemes and discuss the difference between our schemes and the prior `ODXT` under the two-keyword conjunctive queries scenario. We compare the result of our schemes with the only prior art `ODXT` proposed in [25].

6.2 Evaluation Results

Update Time. We respectively run the addition and deletion of our proposed schemes and `ODXT` 10^4 times and measure the average running time. The results are presented in Table 2. From Table 2, it can be seen that the addition time of our proposed schemes is tiny (less than 2 ms), while it is larger than the addition time of `ODXT` (less than 1 ms). This is due to the fact our schemes need to generate the CSRE ciphertexts (see [30] for details) for both "TSet" and "XSet". For deletion, `SDSSE-CQ` and `SDSSE-CQ-S` only need 0.01 ms because they only require 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.

Table 2. Average addition and deletion time of our proposed schemes and `ODXT`

Scheme	`SDSSE-CQ`	`SDSSE-CQ-S`	`ODXT`
Per addition (ms)	1.8	1.8	0.7
Per deletion (ms)	0.01	0.01	0.7

[13] Note that, `ODXT` deploys the technique of `MITRA` [9] to achieve forward privacy, where the client needs to generate a token for each entry in the list of files matching a keyword w.

[14] The modular exponentiation in $\mathbb{G}_T$ is the fastest.

Note that, although our schemes consume more addition time than `ODXT`, our schemes achieve better security and incur less deletion time. Besides, `ODXT` cannot delete a keyword/identifier pair in certain circumstance, because it only add the deletion tag with the corresponding keyword in the "XSet". For example, assume a client deletes a keyword/identifier pair w_2/ind. Then, for a search query $w_1 \wedge w_2$ (assume w_1 is the least frequent keyword and ind contains keyword w_1), the ind will still be returned. In other words, the w_2/ind pair is not deleted.

Search Time. We follow the same setting in [25] to evaluate the schemes. In particular, we execute two types of two-keyword conjunctive queries ($w_1 \wedge w_2$) for the schemes. In the first type of queries, w_1 has a fixed number of corresponding files (i.e., 10), while the number of files matching w_2 varies from 10 to 10^5. Correspondingly, the second type of queries has a fixed number of files (i.e., 10) containing w_2, but the number of files matching w_1 varies from 10 to 10^5.

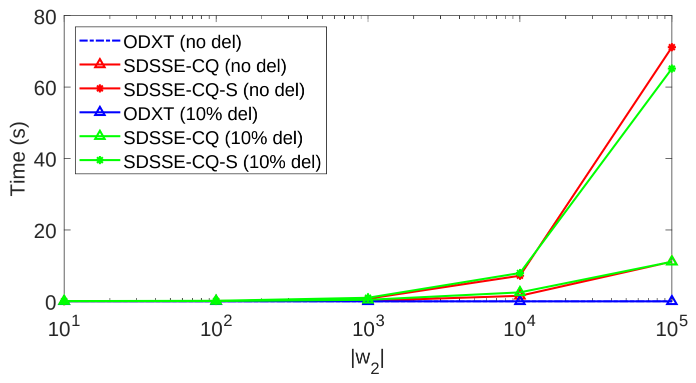

Fig. 2. Search time of `SDSSE-CQ`, `SDSSE-CQ-S` and `ODXT` with constant $|w_1|$

The search time of the first type queries is presented in Figure 2. It shows that the search time increases with the increase of $|w_2|$ for our schemes, while it is almost the same for `ODXT`. This is because `ODXT` only needs to test the same (small) amount of identifiers (i.e., 10) in the "TSet", while our schemes needs to recover the *xtag*s in the "XSet", which are increased with the $|w_2|$. For our schemes, the search time of `SDSSE-CQ` is less than the search time of `SDSSE-CQ-S`. For each scheme, the search time with 10% deletion has a better search time, because the deletion operation reduces the size of the reconstructed "TSet" and "XSet". When $|w_2|$ is small (i.e., 10 to 10^3), the search time of the schemes is almost the same and small. When $|w_2|$ grows larger, our proposed scheme incurs more search time than `ODXT`. Note that, as mentioned before, our schemes achieve better security.

A similar trend can be found in Figure 3, which demonstrates the search time of the second type of queries. The difference is that the search time of

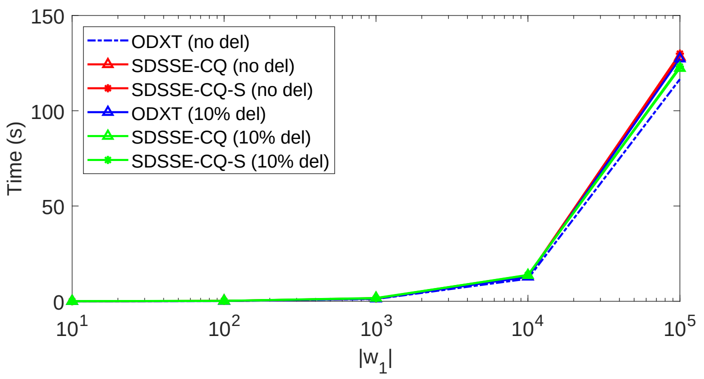

Fig. 3. Search time of `SDSSE-CQ`, `SDSSE-CQ-S` and `ODXT` with constant $|w_2|$

all schemes increases with the increase of $|w_1|$. This is due to the fact that the number of identifiers in "TSet" increases, where each identifier needs to be tested if it contains keyword w_2. This further demonstrates that the influence of $|w_1|$ is dominant in all schemes. Finally, we delete 10% of matched files and evaluate the search time again. We observe that the search time is accelerated after deletion for our schemes, while the search time of `ODXT` is increased. This is because the deletion operation of `ODXT` increases the size of the reconstructed "TSet" and "XSet" ("lazy deletion").

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 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 3, 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". Since the deletion of our schemes only involves operations on the client side, it does not incur any communication cost. For `ODXT`, it has the same communication cost (156 bytes) for both the insertion and deletion process, which is slightly smaller than the communication cost of our schemes.

Table 3. Update communication cost of our proposed schemes and `ODXT` for each keyword/identifier pair

Scheme	`SDSSE-CQ`	`SDSSE-CQ-S`	`ODXT`
Addition (Byte)	172	176	156
Deletion (Byte)	0	0	156

Table 4. Search communication cost of our proposed schemes and `ODXT`

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

Search Communication Cost. Table 4 shows the search communication cost of our proposed schemes and `ODXT`. The search query is two-keyword conjunctive query $(w_1 \wedge w_2)$. The dataset consists of different number of files (i.e., 10^3, 10^4 and 10^5), where each file contains keyword 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 4, 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 contains keyword w_1. Nevertheless, 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 contain keyword w_2.

7 Conclusion

In this paper, we give two DSSE schemes (named `SDSSE-CQ`, `SDSSE-CQ-S`) based on the framework of `OXT` [7]. In addition, we give two different levels of backward privacy for conjunctive queries (named 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. To achieve a stronger level of backward privacy (Type-O$^-$), we give our second scheme (`SDSSE-CQ-S`). The security model of our schemes is more comprehensive for conjunctive queries with `OXT` framework. Moreover, our schemes do not need to send the deleted files to the server, which reduces the interactions between the server and the client. In the future, we would like to make our schemes support more expressive queries.

References

1. G. Amjad, S. Kamara, and T. Moataz, "Breach-resistant structured encryption," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 1, pp. 245–265, 2019.
2. L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks." in *NDSS 2019*. The Internet Society, 2019.
3. R. Bost, "$\Sigma o\varphi o\varsigma$: Forward secure searchable encryption," in *CCS 2016*. ACM, 2016, pp. 1143–1154.

4. R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *CCS 2017*. ACM, 2017, pp. 1465–1482.
5. D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *CCS 2015*. ACM, 2015, pp. 668–679.
6. D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation." in *NDSS 2014*. The Internet Society, 2014.
7. D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO 2013*. Springer, 2013, pp. 353–373.
8. D. Cash and S. Tessaro, "The locality of searchable symmetric encryption," in *EUROCRYPT 2014*. Springer, 2014, pp. 351–368.
9. J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *CCS 2018*. ACM, 2018, pp. 1038–1055.
10. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *CCS 2006*. ACM, 2006, pp. 79–88.
11. A. De Caro and V. Iovino, "jpbc: Java pairing based cryptography," in *ISCC 2011*. IEEE, 2011, pp. 850–855. [Online]. Available: http://gas.dia.unisa.it/projects/jpbc/
12. I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," in *NDSS 2020*. The Internet Society, 2020.
13. I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," in *Annual International Cryptology Conference*. Springer, 2018, pp. 371–406.
14. I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. Garofalakis, and C. Papamanthou, "Practical private range search in depth," *ACM TODS*, vol. 43, no. 1, pp. 2:1–2:52, 2018.
15. M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans, "Efficient dynamic searchable encryption with forward privacy," *Proceedings on Privacy Enhancing Technologies*, vol. 1, pp. 5–20, 2018.
16. S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *ESORICS 2015*. Springer, 2015, pp. 123–145.
17. S. Garg, P. Mohassel, and C. Papamanthou, "Tworam: Efficient oblivious ram in two rounds with applications to searchable encryption," in *Annual Cryptology Conference*. Springer, 2016, pp. 563–592.
18. E.-J. Goh *et al.*, "Secure indexes." *IACR Cryptol. ePrint Arch.*, vol. 2003, p. 216, 2003.
19. P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 315–331.
20. ——, "Learning to reconstruct: Statistical learning theory and encrypted database attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1067–1083.
21. S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *CCS 2012*. ACM, 2012, pp. 965–976.

22. K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1449–1463.
23. S. Lai, S. Patranabis, A. Sakzad, J. K. Liu, D. Mukhopadhyay, R. Steinfeld, S. Sun, D. Liu, and C. Zuo, "Result pattern hiding searchable encryption for conjunctive queries," in *CCS 2018*. ACM, 2018, pp. 745–762.
24. I. Miers and P. Mohassel, "Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality," in *NDSS 2017*. The Internet Society, 2017.
25. S. Patranabis and D. Mukhopadhyay, "Forward and backward private conjunctive searchable symmetric encryption," 2021, https://eprint.iacr.org/2020/1342.
26. D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *S&P 2000*. IEEE, 2000, pp. 44–55.
27. E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *NDSS 2014*, vol. 71. The Internet Society, 2014.
28. E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *CCS 2013*. ACM, 2013, pp. 299–310.
29. S.-F. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen, "An efficient non-interactive multi-client searchable encryption with support for boolean queries," in *ESORICS 2016*. Springer, 2016, pp. 154–172.
30. S.-F. Sun, R. Steinfeld, S. Lai, X. Yuan, A. Sakzad, J. K. Liu, S. Nepal, and D. Gu, "Practical non-interactive searchable encryption with forward and backward privacy," 2021.
31. S.-F. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *CCS 2018*. ACM, 2018, pp. 763–780.
32. J. Wang and S. S. M. Chow, "Forward and backward-secure range-searchable symmetric encryption," *IACR ePrint*, vol. 2019, p. 497, 2019. [Online]. Available: https://eprint.iacr.org/2019/497
33. J. Wang, X. Chen, S. Sun, J. K. Liu, M. H. Au, and Z. Zhan, "Towards efficient verifiable conjunctive keyword search for large encrypted database," in *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 11099. Springer, 2018, pp. 83–100.
34. Y. Wang, J. Wang, S. Sun, M. Miao, and X. Chen, "Toward forward secure SSE supporting conjunctive keyword search," *IEEE Access*, vol. 7, pp. 142 762–142 772, 2019.
35. Z. Wu and K. Li, "Vbtree: forward secure conjunctive queries over encrypted data for cloud computing," *VLDB J.*, vol. 28, no. 1, pp. 25–46, 2019.
36. Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption." in *USENIX-Security 2016*, 2016, pp. 707–720.
37. C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *ESORICS 2019*. Springer, 2019, pp. 283–303.
38. C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and L. Xu, "Forward and backward private dsse for range queries," 2020, p. Early Access.

39. C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption schemes supporting range queries with forward/backward privacy," *CoRR*, vol. abs/1905.08561, 2019. [Online]. Available: http://arxiv.org/abs/1905.08561