Efficient Result-Hiding Searchable Encryption with Forward and Backward Privacy

Takumi Amada[1], Mitsugu Iwamoto[1], and Yohei Watanabe[1,2]

[1] The University of Electro-Communications, Tokyo, Japan
{t.amada, mitsugu, watanabe}@uec.ac.jp
[2] National Institute of Information and Communications Technology, Tokyo, Japan.

Abstract. Dynamic searchable symmetric encryption (SSE) realizes efficient update and search operations for encrypted databases, and there has been an increase in this line of research in the recent decade. Dynamic SSE allows the leakage of insignificant information to ensure efficient search operations, and it is important to understand and identify what kinds of information are insignificant. In this paper, we propose an efficient dynamic SSE scheme Laura under the small leakage, which leads to appealing security requirements such as forward privacy, (Type-II) backward privacy, and result hiding. Laura is constructed based on Aura (NDSS 2021) and is almost as efficient as Aura while only allowing less leakage than Aura. We also provide experimental results to show the concrete efficiency of Laura.

Keywords: Dynamic searchable encryption · Backward Privacy · Encrypted database.

1 Introduction

Searchable symmetric encryption (SSE) [12, 24] provides a way to search a large database efficiently (e.g., cloud storage) for *encrypted* data. In particular, SSE that supports update operations is called *dynamic SSE* [20], which has attracted attention over the past decade [10, 16, 19, 20, 22, 23].

Forward and Backward Privacy. Dynamic SSE aims to efficiently perform keyword searches on encrypted data while revealing some insignificant information to the server. A common understanding of what kinds of leakage are insignificant has been updated by exploring *leakage-abuse attacks* [5, 9, 17, 28] against SSE. In particular, *file injection attacks* demonstrated by Zhang et al. [28] showed that *forward privacy* [7], which guarantees that the adversary cannot learn if newly-added files contain previously-searched keywords, must be a de facto standard security requirement for dynamic SSE.

Backward privacy [8], which guarantees that search operations reveal no useful information on previously-deleted files even if they contain searched keywords, has been spotlighted since it sounds like another natural security requirement. However, it is more difficult to achieve backward privacy than forward privacy since it is just like we require the server to forget previously-stored information. For example, we have to hide even information about when and which files are added and/or deleted to meet backward privacy. Therefore, one of the current major research interests in dynamic SSE is how efficiently we construct dynamic SSE schemes with backward privacy.

Importance of Result-Hiding SSE. As described above, leakage-abuse attacks tell us which information should not be leaked during update and search operations. Existing attacks are classified into *passive* and *active* ones. Passive attacks (e.g., [17]) aim to identify keywords behind search queries from admitted leakage information and seem more likely to happen in the real world than active attacks (e.g., [28]), which require that the server can force the client to upload arbitrary files. A major drawback of passive attacks is that they also require partial information of the stored data as extra information in addition to the leakage profiles. This is quite an unrealistic assumption. Hence, the subsequent works (e.g., [5, 9]) have attempted to weaken the assumption. Recently, Bkackstone et al. [5] showed passive attacks that only require 5% of the client's data, whereas the Islam et al.'s seminal work [17] requires at least 95% of the client's data. In particular, it is worth noting that their attacks only use *access pattern leakage*, which is a standard leakage profile of dynamic SSE. Although there are, fortunately, countermeasures such as volume-hiding techniques [18], they significantly decrease the efficiency of dynamic SSE schemes. Thus, it becomes more important to seek efficient constructions of *result-hiding schemes*, which are dynamic SSE schemes mitigating access pattern leakage.

1.1 Our Contribution

In this paper, we propose Laura, a new result-hiding dynamic SSE scheme with forward and Type-II backward privacy, which is the most investigated security level of backward privacy. Laura is constructed based on Aura [25]; Laura is built from only symmetric-key primitives, specifically, from any pseudorandom function (PRF), any symmetric-key encryption (SKE), and any approximate membership query (AMQ) data structure. Laura achieves better practical efficiency to Aura and requires less leakage than Aura; this is the reason why we call our scheme Laura, which stands for Low-leakage Aura.

We give experimental results to show the concrete efficiency of Laura and v-Laura, which is a variant of Laura; their deletion and search procedures are almost as efficient as Aura, and their addition procedures are substantially more efficient than Aura. For example, Laura and v-Laura take less than a second to add 200,000 entries, while Aura takes about a minute. For concrete efficiency comparison among Laura, v-Laura, and Aura, see Section 6.

As a side result, we also figure out that in Aura (as well as Laura and v-Laura), the client is assumed to never re-add any keyword-identifier pair (w, id) once deleted, where id is a file identifier. This assumption seems reasonable in practice since id should be replaced with a new one if the client wants to re-add a previously-deleted file whose identifier is id. We also show a variant of Laura,

Table 1: Efficiency comparison among Type-II backward-private dynamic SSE schemes. Suppose that the client has performed search and update operations t times in total. d and n are the total numbers of distinct keywords and files, respectively. a_w, n_w, and $n_{w,\mathsf{del}}^{(\mathtt{srch})}$ are the total numbers of all updates for w, files currently containing a keyword w, and times a keyword w has been affected by search operations since the last search for w, respectively. It clearly holds $a_w \geq \widehat{n}_w \geq n_w$, where $\widehat{n}_w := n_w + n_{w,\mathsf{del}}^{(\mathtt{srch})}$. N is the total numbers of (document, keyword) pairs, i.e., $N := \Sigma_w n_w$. Let $N' := \Sigma_w \widehat{n}_w$ and $\widehat{N} := \Sigma_w a_w$. Namely, it holds $\widehat{N} \geq N' \geq N$. $|\sigma|$ and $|\mathsf{EDB}|$ denote bit-lengths of client's state information and encrypted database. RT and RH stand for roundtrips and result hiding, respectively. SK indicates whether the scheme is constructed from only symmetric-key primitives. RU stands for re-updatability, which allows the client to re-add a previously-deleted entry (w, id) to EDB.

	$\|\sigma\|$	$\|\mathsf{EDB}\|$	Update		Search		RT	RH	SK	RU
			Comp.	Comm.	Comp.	Comm.				
SD_a [13]	$\mathcal{O}(1)$	$\mathcal{O}(\widehat{N})$	$\mathcal{O}(\log \widehat{N})^\dagger$	$\mathcal{O}(\log \widehat{N})^\dagger$	$\mathcal{O}(\widehat{a}_w)^\sharp$	$\mathcal{O}(\widehat{n}_w)$	2	✓	✓	✓
SD_d [13]	$\mathcal{O}(1)$	$\mathcal{O}(\widehat{N})$	$\mathcal{O}(\log^3 \widehat{N})$	$\mathcal{O}(\log \widehat{N})$	$\mathcal{O}(\widehat{a}_w)^\sharp$	$\mathcal{O}(n_w)$	2	✓	✓	✓
Fides [8]	$\mathcal{O}(d)$	$\mathcal{O}(N')$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\widehat{n}_w)$	$\mathcal{O}(\widehat{n}_w)$	3	✓	—	✓
Aura [25] (+EKPE [14])	$\mathcal{O}(d)$	$\mathcal{O}(\widehat{N})$	$\mathcal{O}(1)^\ddagger$	$\mathcal{O}(1)$	$\mathcal{O}(\widehat{n}_w)$	$\mathcal{O}(n_w)$	1	—	✓	—
Laura (§4.2)	$\mathcal{O}(d)$	$\mathcal{O}(N')$	$\mathcal{O}(1)^\ddagger$	$\mathcal{O}(1)$	$\mathcal{O}(\widehat{n}_w)$	$\mathcal{O}(\widehat{n}_w)$	3	✓	✓	—
v-Laura (§5.1)	$\mathcal{O}(d)$	$\mathcal{O}(N')$	$\mathcal{O}(1)^\ddagger$	$\mathcal{O}(1)$	$\mathcal{O}(\widehat{n}_w)$	$\mathcal{O}(\widehat{n}_w)$	2	✓	✓	—
s-Laura (§5.2)	$\mathcal{O}(d)$	$\mathcal{O}(N')$	$\mathcal{O}(1)^\ddagger$	$\mathcal{O}(1)$	$\mathcal{O}(\widehat{n}_{w,\mathsf{del}}^{(\mathtt{srch})})^\sharp$	$\mathcal{O}(\widehat{n}_w)$	3	✓	✓	✓

† Amortized analysis.

‡ To be precise, the deletion procedure depends on the time complexity of the underlying AMQ structure, which is $\mathcal{O}(1)$ in almost all existing constructions.

$^\sharp$ Let $\widehat{a}_w := a_w + \log \widehat{N}$ and $\widehat{n}_{w,\mathsf{del}}^{(\mathtt{srch})} := \widehat{n}_w \cdot n_{w,\mathsf{del}}^{(\mathtt{srch})}$ for compact notations.

called s-Laura, that removes the assumption, i.e., it allows the client to re-add previous-deleted entries to the encrypted database, although s-Laura requires extra search costs.

Efficiency Comparison. We compare the asymptotic efficiency of dynamic SSE schemes with forward and Type-II backward privacy in Table 1. Note that we evaluate the server-side complexities of update and search algorithms. Although the efficiency of Laura and v-Laura seems comparable to Fides [8] and Aura [25], Laura and v-Laura has clear advantages over them; Fides employs public-key primitives such as trapdoor permutations for its building block. Moreover, Fides returns a (tentative) search result that contains deleted identifiers. Therefore, the client themself has to remove such deleted ones to obtain the correct search result. Although Laura and v-Laura also require for the client to remove deleted identifiers, the client can easily find them thanks to the underlying approximate membership query (AMQ) data structure. Aura achieves the minimum roundtrip, however, the size of encrypted databases is large. Furthermore, Aura reveals the access pattern and therefore is *not* a result-hiding scheme. Among the dynamic SSE schemes that satisfy all properties (RH, SK, and RU) listed in the table, $\mathsf{s\text{-}Laura}$ is more efficient than SD_a and SD_d.

2 Preliminaries

Notations. For any integer $a \in \mathbb{Z}$, let $[a] \coloneqq \{1, 2, \ldots, a\}$. For a finite set $\mathcal{X}$, we use $x \xleftarrow{\$} \mathcal{X}$ to represent processes of choosing an element x from X uniformly at random. For a finite set $\mathcal{X}$, we denote by $\mathcal{X} \leftarrow x$ and $|\mathcal{X}|$ the addition x to $\mathcal{X}$ and cardinality of $\mathcal{X}$, respectively. Concatenation is denoted by $\|$. In the description of the algorithm, all arrays, strings, and sets are initialized to empty ones. We consider probabilistic polynomial time (PPT) algorithms. For any non-interactive algorithm A, $\mathsf{out} \leftarrow \mathsf{A}(\mathsf{in})$ means that A takes in as input and outputs out. In this paper, we consider two-party interactive algorithms between a client and a server, and it is denoted by $(\mathsf{out}_\mathsf{C}; \mathsf{out}_\mathsf{S}) \leftarrow \mathsf{A}(\mathsf{in}_\mathsf{C}; \mathsf{in}_\mathsf{S})$, where in_C and in_S are input of client and server, respectively and out_C and out_S are output of client and server, respectively. If necessary, we mention the transcript trans and describe the algorithm as $\langle(\mathsf{out}_\mathsf{C}; \mathsf{out}_\mathsf{S}),\ \mathrm{trans}\rangle \leftarrow \mathsf{A}(\mathsf{in}_\mathsf{C}; \mathsf{in}_\mathsf{S})$. The security parameter and negligible function are denoted by κ and $\mathsf{negl}(\cdot)$, respectively.

Pseudorandom Functions (PRFs). A family of functions $\pi \coloneqq \{\pi_{\mathsf{k}_{\mathrm{PRF}}} : \{0,1\}^* \to \{0,1\}^m\}_{\mathsf{k}_{\mathrm{PRF}} \in \{0,1\}^\kappa}$, where $m = \mathsf{poly}(\kappa)$, is said to be a (variable-input-length) PRF family if for sufficiently large $\kappa \in \mathbb{N}$ and all PPT algorithm D, it holds $|\Pr[\mathsf{D}^{\pi(\mathsf{k}_{\mathrm{PRF}},\cdot)}(1^\kappa) = 1 \mid \mathsf{k}_{\mathrm{PRF}} \xleftarrow{\$} \{0,1\}^\kappa] - \Pr[\mathsf{D}^{\mathrm{R}(\cdot)}(1^\kappa) = 1 \mid \mathrm{R} \xleftarrow{\$} \mathcal{R}]| < \mathsf{negl}(\kappa)$, where $\mathcal{R}$ is a set of all mappings $\mathrm{R} : \{0,1\}^* \to \{0,1\}^m$.

Symmetric-Key Encryption (SKE). An SKE Π_{SKE} consists of three PPT algorithms $\Pi_{\mathrm{SKE}} = (\mathsf{G}, \mathsf{E}, \mathsf{D})$. G takes a security parameter κ as input and outputs a secret key $\mathsf{k}_{\mathrm{SKE}}$, and E takes a plaintext m and $\mathsf{k}_{\mathrm{SKE}}$ as input and outputs the ciphertext c. D takes c with $\mathsf{k}_{\mathrm{SKE}}$ and outputs m or $\perp$ as a symbol of failure. In this paper, we assume Π_{SKE} is CPA security. For formal definitions, we refer the readers to [21]. Also, if necessary, we explicitly describe a nonce used in an SKE. Specifically, for nonce r, the encryption and decryption algorithms are denoted by $\mathsf{E}(\mathsf{k}_{\mathrm{SKE}}, \mathsf{m}; r)$ and $\mathsf{D}(\mathsf{k}_{\mathrm{SKE}}, \mathsf{c}; r)$, respectively. The ciphertext is treated as $r\|\mathsf{c}$. Note that (nonce-based) CTR and CBC modes in block ciphers satisfy CPA security and the above properties.

Approximate Membership Query (AMQ) Structure. Probabilistic data structures, known as Approximate Membership Query (AMQ) data structures, provide membership queries with compact data sizes by allowing "false positives." The most appealing feature of AMQ structures is to make the false-positive probability small enough by setting specific parameters appropriately. We consider AMQ structures that support both insertion and deletion operations. While the Bloom filter [6], one of the well-known AMQ structures, does not support deletion, recent ones, such as the cuckoo filter [15] and quotient filter [4], do. Formally, an arbitrary set $\mathcal{U} \in \{0,1\}^*$, an AMQ data structure $\Pi_{\text{AMQ}} = (\mathsf{AMQ.Gen}, \mathsf{AMQ.Insert}, \mathsf{AMQ.Delete}, \mathsf{AMQ.Lookup})$ consists of the following PPT algorithms:

– $(\mathcal{T}, \mathsf{aux}) \leftarrow \mathsf{AMQ.Gen}(\mathcal{U}, \mathsf{par})$: it takes $\mathcal{U}$ and a parameter par as input, and outputs an initial structure $\mathcal{T}$ and auxiliary information aux. The parameter par depends on the construction of the specific AMQ structure.
– $\mathcal{T}' \leftarrow \mathsf{AMQ.Insert}(\mathcal{T}, x, \mathsf{aux})$: it takes as input a data structure $\mathcal{T}$, an element $x \in \mathcal{U}$ to be added, and aux, and outputs an updated structure $\mathcal{T}'$.
– $\mathcal{T}' \leftarrow \mathsf{AMQ.Delete}(\mathcal{T}, x, \mathsf{aux})$: it takes as input a data structure $\mathcal{T}$, an element $x \in \mathcal{U}$ to be deleted, and aux, and outputs an updated structure $\mathcal{T}'$.
– $\mathtt{true}/\mathtt{false} \leftarrow \mathsf{AMQ.Lookup}(\mathcal{T}, x, \mathsf{aux})$: it takes as input a data structure $\mathcal{T}$, an element $x \in \mathcal{U}$ to be queried, and aux, and outputs $\mathtt{true}$ or $\mathtt{false}$.

AMQ structures meet the following two properties. Due to the page limitation, we omit the formal description and will give it in the full version.

– *Completeness*: Let $\mathcal{S}$ be a set of elements that have been inserted (and not deleted). For all $x \in \mathcal{S}$, it holds $\mathsf{AMQ.Lookup}(\mathcal{T}, x, \mathsf{aux}) = \mathtt{true}$, where $\mathcal{T}$ is the corresponding structure.
– *Bounded False-Positive Probability*: Let $n := |\mathcal{S}|$. Then, there exists $\mu_n \in (0,1]$ such that it holds $\Pr[\mathsf{AMQ.Lookup}(\mathcal{T}, x, \mathsf{aux}) = \mathtt{true}] \leq \mu_n$ for any $x \in \mathcal{U} \setminus \mathcal{S}$.

3 Dynamic SSE

3.1 Notation for Dynamic SSE

$\Lambda := \{0,1\}^\lambda$ is a set of possible keywords (sometimes called a *dictionary*), where $\lambda = \mathsf{poly}(\kappa)$. A document f_{id} has its unique identifier $\mathsf{id} \in \{0,1\}^\ell$, which is irrelevant to the contents of f_{id}, where $\ell = \mathsf{poly}(\kappa)$. A counter t represents the global counter through the protocol; it is initialized to 0 at setup and incremented for each search or update operation. A database $\mathsf{DB}^{(t)}$ at t is represented as a set of keyword-identifier pairs (w, id), i.e., $\mathsf{DB}^{(t)} := \{(w_i, \mathsf{id}_i)\}_{i=1}^{N(t)}$, where $N(t)$ is the number of pairs stored in the server at t. We denote $\mathsf{ID}^{(t)}$ by a set of identifiers in $\mathsf{DB}^{(t)}$. That is, $\mathsf{ID}^{(t)} := \{\mathsf{id} \mid \forall w \in \Lambda, (w, \mathsf{id}) \in \mathsf{DB}^{(t)}\}$. Similarly, $\mathcal{W}^{(t)}$ is denoted by a set of keywords in $\mathsf{DB}^{(t)}$, i.e., $\mathcal{W}^{(t)} := \{w \mid \forall \mathsf{id} \in \mathsf{ID}^{(t)}, (w, \mathsf{id}) \in \mathsf{DB}^{(t)}\}$.

3.2 Model

Dynamic SSE consists of three PPT algorithms $(\mathsf{Setup}, \mathsf{Update}, \mathsf{Search})$. Firstly, the client runs Setup to generate a secret key, initial state information, and an initial encrypted database, which is sent to the server. The client interacts with the server and runs Update and Search repeatedly to add or delete a pair (w, id) and search for keywords.

Definition 1 (Dynamic SSE). *A Dynamic SSE* $\Sigma := (\mathsf{Setup}, \mathsf{Update}, \mathsf{Search})$ *over* Λ *consists of the following PPT algorithms:*

- $(k, \sigma^{(0)}, \mathsf{EDB}^{(0)}) \leftarrow \mathsf{Setup}(1^\kappa)$*: it is an non-interactive algorithm that takes a security parameter* κ *as input and outputs a secret key* k*, initial state information* $\sigma^{(0)}$*, and initial encrypted database* $\mathsf{EDB}^{(0)}$*.*
- $(\sigma^{(t+1)}; \mathsf{EDB}^{(t+1)}) \leftarrow \mathsf{Update}(k, \mathsf{op}, \mathsf{in}, \sigma^{(t)}; \mathsf{EDB}^{(t)})$*: it is an interactive algorithm that takes* k*, an operation label* $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$*, the corresponding input* $\mathsf{in} := (w, \mathsf{id})$*, and* $\sigma^{(t)}$ *as input of the client and encrypted database* $\mathsf{EDB}^{(t)}$ *as input of the server, and outputs updated state information* $\sigma^{(t+1)}$ *for the client and updated encrypted database* $\mathsf{EDB}^{(t+1)}$ *for the server.*
- $(\mathcal{X}_q^{(t)}, \sigma^{(t+1)}; \mathsf{EDB}^{(t+1)}) \leftarrow \mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$*: it is an interactive algorithm that takes* k*, a searched keyword* q*, and* $\sigma^{(t)}$ *as input of the client and encrypted database* $\mathsf{EDB}^{(t)}$ *as input of the server, and outputs updated state information* $\sigma^{(t+1)}$ *and a search result* $\mathcal{X}_q^{(t)}$ *for the client and updated encrypted database* $\mathsf{EDB}^{(t+1)}$ *for the server.*

Briefly, the correctness of the above model ensures that it holds $\mathcal{X}_q^{(t)} = \{\mathsf{id} \in \mathsf{ID}^{(t)} \mid (q, \mathsf{id}) \in \mathsf{DB}^{(t)}\}$ with overwhelming probability for any keyword $q \in \Lambda$. For a formal definition, we refer the readers to [10].

3.3 Security

Dynamic SSE guarantees that the (honest-but-curious) server does not learn any information beyond some explicit information leakage during a sequence of operations. Therefore, such information leakage is characterized as a leakage function $\mathcal{L} := (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Upd}}, \mathcal{L}_{\mathsf{Srch}})$, where $\mathcal{L}_{\mathsf{Setup}}$, $\mathcal{L}_{\mathsf{Upd}}$, and $\mathcal{L}_{\mathsf{Srch}}$ are functions that refer to information leaked during Setup, Update, and Search, respectively.

$\mathcal{L}$-Adaptive Security. We define $\mathcal{L}$-adaptive security of SSE in a simulation-based manner. We consider two experiments: a real experiment Real in which the Dynamic SSE scheme is performed in the real world and an ideal experiment Ideal that at most leaks a leakage function $\mathcal{L}$. Specifically, a real experiment Real_D is performed by the client and a PPT algorithm $\mathsf{D} = (\mathsf{D}_1, \mathsf{D}_2, \ldots, \mathsf{D}_{Q+1})$, while ideal experiment $\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}$ is performed by D and a simulator $\mathsf{S} = (\mathsf{S}_0, \mathsf{S}_1, \ldots, \mathsf{S}_Q)$ with leakage function $\mathcal{L}$. In each experiment, D adaptively queries and attempts to distinguish between the two experiments. If D cannot distinguish between them, D has not learned more information than the leakage function $\mathcal{L}$; we call this $\mathcal{L}$-adaptive security. Each experiment is formally given in Fig. 1, and the security definition is as follows [27].

Definition 2 ($\mathcal{L}$-Adaptive Security). *Let* Σ *be a Dynamic SSE scheme.* Σ *is* $\mathcal{L}$*-adaptively secure, with regard to a leakage function* $\mathcal{L}$*, if for any PPT algorithm* D*, any sufficiently large* $\kappa \in \mathbb{N}$*, and any* $Q := \mathsf{poly}(\kappa)$*, there exists a PPT algorithm* S *s.t.* $|\Pr[\mathsf{Real}_\mathsf{D}(\kappa, Q) = 1] - \Pr[\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}(\kappa, Q) = 1]| < \mathsf{negl}(\kappa)$.

Real Experiment: $\mathsf{Real}_{\mathsf{D}}(\kappa, Q)$

```
1: $(k, \sigma^{(0)}, \mathsf{EDB}^{(0)}) \leftarrow \mathsf{Setup}(1^\kappa)$
2: $\mathsf{st_D} := \{\mathsf{EDB}^{(0)}\}$
3: for $t = 1$ to $Q$ do
4:     $\mathsf{query} \leftarrow \mathsf{D}_t(\mathsf{st_D})$
5:     if $\mathsf{query} = (\mathtt{upd}, \mathsf{op}, \mathsf{in})$ then
6:         $\langle(\sigma^{(t)}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle$
           $\leftarrow \mathsf{Update}(k, \mathsf{op}, \mathsf{in}, \sigma^{(t-1)}; \mathsf{EDB}^{(t-1)})$
7:     if $\mathsf{query} = (\mathtt{srch}, q)$ then
8:         $\langle(\sigma^{(t)}, \mathcal{X}_q^{(t-1)}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle$
           $\leftarrow \mathsf{Search}(k, q, \sigma^{(t-1)}; \mathsf{EDB}^{(t-1)})$
9:     $\mathsf{st_D} \leftarrow (\mathsf{EDB}^{(t)}, \mathsf{trans}^{(t)})$
10: $b \leftarrow \mathsf{D}_{Q+1}(\mathsf{st_D})$
11: return $b$
```

Ideal Experiment: $\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}(\kappa, Q)$

```
1: $(\mathsf{EDB}^{(0)}, \mathsf{st_S}) \leftarrow \mathsf{S}_0(\mathcal{L}_{\mathsf{Setup}}(\kappa))$
2: $\mathsf{st_D} := \{\mathsf{EDB}^{(0)}\}$
3: for $t = 1$ to $Q$ do
4:     $\mathsf{query} \leftarrow \mathsf{D}_t(\mathsf{st_D})$
5:     if $\mathsf{query} = (\mathtt{upd}, \mathsf{op}, \mathsf{in})$ then
6:         $\langle(\mathsf{st}'_\mathsf{S}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle$
           $\leftarrow \mathsf{S}_t(\mathsf{st_S}, \mathcal{L}_{\mathsf{Upd}}(t, \mathsf{op}, \mathsf{in}); \mathsf{EDB}^{(t-1)})$
7:     if $\mathsf{query} = (\mathtt{srch}, q)$ then
8:         $\langle(\mathsf{st}'_\mathsf{S}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle$
           $\leftarrow \mathsf{S}_t(\mathsf{st_S}, \mathcal{L}_{\mathsf{Srch}}(t, q); \mathsf{EDB}^{(t-1)})$
9:     $\mathsf{st_D} \leftarrow (\mathsf{EDB}^{(t)}, \mathsf{trans}^{(t)})$
10:    $\mathsf{st_S} := \mathsf{st}'_\mathsf{S}$
11: $b \leftarrow \mathsf{D}_{Q+1}(\mathsf{st_D})$
12: return $b$
```

Fig. 1: Real and ideal experiments.

Forward and Backward Privacy. The well-known security notions for update operations are *forward privacy* [7] and *backward privacy* [8].

Forward privacy, roughly speaking, ensures that while running an update of a keyword-identifier pair (q, id), no information about the keyword q is exposed to the server. This means that the keyword q cannot be associated with all previous searches and update operations. Forward privacy is an important security requirement since Zhang et al. [28] showed effective attacks against non-forward-private dynamic SSE schemes. The formal definition is as follows:

Definition 3 (Forward Privacy [7]). *Let Σ be a $\mathcal{L}$-adaptively secure dynamic SSE scheme. Σ is forward private if $\mathcal{L}_{\mathsf{Upd}}$(for $\mathsf{op} = \mathsf{add}$) can be written as $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{add}, (q, \mathsf{id})) = \mathcal{L}'(t, \mathsf{add}, \mathsf{id})$, where $\mathcal{L}'$ is stateless function.*

On the other hand, loosely speaking, backward privacy guarantees that while running a search for a keyword q, the least possible (ideally, no) information about the deleted pair (q, id) is leaked to the server. However, if leakage regarding deletion operations is to be completely eliminated, significant costs are required due to efficiency trade-offs. Therefore, Bost et al. [8] introduced three levels of backward privacy: from Type-I with the least leakage to Type-Ⅲ with the most leakage. In this paper, we focus on Type-Ⅱ backward privacy, which achieves a good balance between security levels and achievable efficiency. To describe their definition, we define several functions of leaked information as follows. Let $\mathcal{Q}^{(t)}$ be the set of all operations of each counter $u \in [t]$, and its elements are described as $(u, q) \in \mathcal{Q}^{(t)}$ for a search for a keyword q and $(u, \mathsf{op}, (q, \mathsf{id}))) \in \mathcal{Q}^{(t)}$ for an update of a keyword-identifier pair (q, id), where $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$.

- **Search pattern** $\mathsf{SP}_q^{(t)}$: A set of counters at which the keyword q has been searched. That is, $\mathsf{SP}_q^{(t)} := \{u \mid (u, q) \in \mathcal{Q}^{(t)}\}$.
- **Access pattern** $\mathsf{TimeDB}_q^{(t)}$: A set of pairs of an identifier $\mathsf{id} \in \mathsf{ID}^{(t)}$ that includes a keyword q at t and a counter u when the corresponding keyword-identifier pair (q, id) was added. That is,

$$\mathsf{TimeDB}_q^{(t)} := \left\{ (u^{\mathsf{add}}, \mathsf{id}) \;\middle|\; \begin{array}{l} (u^{\mathsf{add}}, \mathsf{add}, (q, \mathsf{id})) \in \mathcal{Q}^{(t)} \\ \wedge\ \forall u^{\mathsf{del}}, (u^{\mathsf{del}}, \mathsf{del}, (q, \mathsf{id})) \notin \mathcal{Q}^{(t)} \end{array} \right\},$$

where we assume $u^{\mathsf{add}} < u^{\mathsf{del}}$ without the loss of generality.
- **Update pattern** $\mathsf{Update}_q^{(t)}$: It is a set of counters for all update operations on q, i.e., $\mathsf{Update}_q^{(t)} := \big\{u \mid (u, \mathsf{add}, (q, \mathsf{id})) \in \mathcal{Q}^{(t)} \ \vee\ (u, \mathsf{del}, (q, \mathsf{id})) \in \mathcal{Q}^{(t)}\big\}$.

Using the above functions, Type-II backward privacy is defined as follows.

Definition 4 (Type-II Backward Privacy [8]). *Let Σ be a $\mathcal{L}$-adaptively secure dynamic SSE scheme. Σ is Type-II backward private if $\mathcal{L}_{\mathsf{Upd}}$ and $\mathcal{L}_{\mathsf{Srch}}$ can be written as:*

$$\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{op}, (q, \mathsf{id})) = \mathcal{L}'(t, \mathsf{op}, q) \ \text{and}\ \mathcal{L}_{\mathsf{Srch}}(t, q) = \mathcal{L}''(\mathsf{SP}_q^{(t)}, \mathsf{TimeDB}_q^{(t)}, \mathsf{Update}_q^{(t)}),$$

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

Result Hiding. As mentioned in the introduction, taking into account the recent progress in leakage abuse attacks, it is important to realize an efficient dynamic SSE scheme that never leaks identifiers of search results. Such a scheme is called a *result-hiding* one. Although several result-hiding schemes [13, 8] are already known, to the best of our knowledge, there is no formal definition of the result-hiding property. Therefore, we first define it formally. We consider the following leakage functions.

- **Concealed access pattern** $\mathsf{Time}_q^{(t)}$: It is a set of counters contained in $\mathsf{TimeDB}_q^{(t)}$. That is, $\mathsf{Time}_q^{(t)} := \{u \mid \exists \mathsf{id} \text{ s.t. } (u, \mathsf{id}) \in \mathsf{TimeDB}_q^{(t)}\}$.
- **Deletion history** $\mathsf{DelHist}_q^{(t)}$: It is a set of pairs of two counters at which each of addition and deletion operations is performed on the same (q, id) pair. That is,

$$\mathsf{DelHist}_q^{(t)} := \left\{ (u^{\mathsf{add}}, u^{\mathsf{del}}) \;\middle|\; \begin{array}{l} \exists \mathsf{id} \text{ s.t. } (u^{\mathsf{add}}, \mathsf{add}, (q, \mathsf{id})) \in \mathcal{Q}^{(t)} \\ \wedge\ (u^{\mathsf{del}}, \mathsf{del}, (q, \mathsf{id})) \in \mathcal{Q}^{(t)} \end{array} \right\}.$$

Although $\mathsf{DelHist}_q^{(t)}$ is a well-known leakage function to define Type-III backward privacy, we also use it to define the result-hiding property.

Definition 5 (Result-Hiding Dynamic SSE). *Let Σ be a $\mathcal{L}$-adaptively secure dynamic SSE scheme. Σ is called a result-hiding scheme if $\mathcal{L}_{\mathsf{Upd}}$ and $\mathcal{L}_{\mathsf{Srch}}$ can be written as:*

$$\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{op}, (q, \mathsf{id})) = \mathcal{L}'(t, \mathsf{op}, q) \ \text{and}\ \mathcal{L}_{\mathsf{Srch}}(t, q) = \mathcal{L}''(\mathsf{SP}_q^{(t)}, \mathsf{Time}_q^{(t)}, \mathsf{DelHist}_q^{(t)}),$$

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

Namely, result-hiding schemes do not leak any identifiers during updates and searches. Note that the search operation may leak all information related to the counters of update operations on q since the result-hiding property should be a property in which result-hiding schemes reveal no information on identifiers *themselves* contained in search results.

Remark 1. One may think that the result-hiding property conflicts with a common use case of dynamic SSE, where the server returns both a search result and the corresponding actual documents. The property prevents the server from returning the actual documents unless the client reveals the search result to the server; the reveal means the leakage of the access pattern and makes the result-hiding property meaningless! Nevertheless, in such a common use case, the result-hiding property should be valuable since the client can choose whether the client reveals the access pattern. Of course, the property would be more appealing in other use cases, e.g., where actual documents are stored on another server.

4 Laura: Low-leakage Aura

We propose a new efficient dynamic SSE scheme that meets forward privacy, Type-II backward privacy, and the result-hiding property. Although the construction approach of our scheme is based on Aura, our scheme allows less leakage than Aura. Thus, we call our scheme Laura, which stands for low-leakage Aura.

4.1 Construction Idea

Construction Overview of Aura. Sun et al. [25] introduced a core building block of Aura, called symmetric revocable encryption (SRE). Briefly speaking, SRE supports *puncturable decryption*. In SRE, plaintexts are encrypted along with a tag. A decryption key associated with a certain revoked set, containing revoked tags, cannot decrypt ciphertexts related to the revoked tags. In Aura, SRE's puncturable decryption functionality allows the server to decrypt ciphertexts without leaking deleted entires as follows. When adding (w, id), the client encrypts id with a tag τ and the ciphertext is stored on the server. When deleting (w, id), the client adds the corresponding tag τ to a revoked tag set $\mathcal{R}_w$ on w, stored in the local storage. When searching for w, the client retrieves the revoked tag set $\mathcal{R}_w$ and generates a decryption key associated with $\mathcal{R}_w$. The server decrypts ciphertexts with the key and obtains id if the corresponding tag τ' is not revoked (i.e., $\tau' \notin \mathcal{R}_w$); it obtains $\perp$ otherwise due to the puncturable decryption functionality. Therefore, the client can delegate the process of removing deleted entries to the server; it does not leak when and which identities have been deleted. The client just receives and outputs the search result from the server. Consequently, Aura is the first (efficient) dynamic SSE that supports both non-interactive search operations and Type-II backward privacy. However, there is still room for improvement as follows:

1) Although a Bloom filter [6] is used to compress the revoked tag set $\mathcal{R}_w$, the client has to store them on the local storage. It is desirable to reduce the amount of local storage on the client side (i.e., state information) as much as possible.
2) Aura employs *logical deletion*; for a deletion operation of (w, id), an entry $(\mathsf{del}, (w, \mathsf{id}))$ is added to an encrypted database EDB. As a result, the size of EDB in Aura is $\widehat{N} = \sum_w a_w$, where a_w is the total number of updates for w.
3) As seen above, the server decrypts the ciphertexts and gets the access patterns. Namely, Aura is not a result-hiding scheme.

Our Approach. A common approach to realizing result-hiding schemes is to have the client decrypt the search results [8, 11, 13]. With this approach in mind, our scheme is based on Aura combined with Etemad et al.'s forward-private scheme [14], which are *not* result-hiding schemes; we no longer employ SRE but the concept of revoked tags. The construction idea for Laura is to perform a *variant of* logical deletion using tags; sending the server a revoked tag τ of a deleted pair (w, id), instead of the (encrypted) pair itself, when deleting (w, id). Therefore, the client does not have to remember the tags. Laura maintains the revoked tags with an (arbitrary) AMQ data structure that supports deletion operations, whereas Sun et al. [25] only considered the Bloom filter for Aura. Hence, the client easily finds the deleted entries with the $\mathsf{AMQ.Lookup}$ algorithm, which leads to the result-hiding property while keeping efficiency.[3]

Moreover, we also achieve a smaller EDB through re-addition techniques [14, 26]: for a search query on w, the server retrieves all values related to the query from EDB and deletes them. After getting the search result, the client re-adds all entries except for deleted ones for the next search. We summarize what our approach resolves.

1) Laura achieves a smaller (concrete) storage size on the client side than Aura.
2) Laura achieves a smaller $|\mathsf{EDB}| = N' = \sum_w (n_w + n_{w,\mathsf{del}}^{(\mathtt{srch})})$ than Aura, where $n_{w,\mathsf{del}}^{(\mathtt{srch})}$ is the total number of times a keyword w has been affected by search operations since the last search for w. It clearly holds $\widehat{N} \geq N'$.
3) Laura is a result-hiding scheme. Furthermore, compared to existing these schemes, Laura achieves compression of EDB and efficient removal of deleted entries due to the AMQ data structure.

In addition to the above benefits, Laura is more practically efficient than Aura. We will see that in Section 6.

[3] Though the server needs to send the AMQ structure to the client during the search operation, the size of the structure is reasonably small. For example, if we select the cuckoo filter [15] as the AMQ structure, its size is 0.79 MB for 100,000 deleted entries with the false-positive probability $p = 10^{-4}$. As a reference, according to the Aura paper [25], SD_d [13] requires 8.58 MB of total communication costs for search.

```
Algorithm: Laura
Setup(1^κ)
Client:
 1: $\mathsf{k}_{\mathrm{PRF}}, \mathsf{k}_{\mathrm{RH}}, \mathsf{k}_{\mathrm{SKE}} \xleftarrow{\$} \{0,1\}^\kappa$
 2: $(\mathcal{T}, \mathsf{aux}) \leftarrow \mathsf{AMQ.Gen}(\{0,1\}^\lambda, \mathsf{par})$
 3: $\mathsf{fc}_w, \mathsf{sc}_w, \mathsf{Index}[] := \varepsilon$ // $\varepsilon$ is an empty value
 4: return $\Big(k := (\mathsf{k}_{\mathrm{PRF}}, \mathsf{k}_{\mathrm{RH}}, \mathsf{k}_{\mathrm{SKE}}), \sigma^{(0)} := (\mathsf{sc}_w, \mathsf{fc}_w), \mathsf{EDB}^{(0)} := (\mathsf{Index}, \mathcal{T}, \mathsf{aux})\Big)$

Update$(k, \mathsf{add}, (w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$
Client:
 1: $\tau \leftarrow \pi(\mathsf{k}_{\mathrm{RH}}, w \| \mathsf{id})$
 2: if $\mathsf{sc}_w$ is undefined then
 3:     $(\mathsf{sc}_w, \mathsf{fc}_w) := (0, 0)$
 4: $\mathsf{fc}_w := \mathsf{fc}_w + 1$ // increment $\mathsf{fc}_w$
 5: $\mathsf{K}_w^{(\mathsf{sc}_w)} \leftarrow g(\mathsf{k}_{\mathrm{PRF}}, w \| \mathsf{sc}_w)$ // generate the PRF key for address
 6: $\mathtt{addr} \leftarrow h(\mathsf{K}_w^{(\mathsf{sc}_w)}, \mathsf{fc}_w)$,    $\mathtt{val} \leftarrow \mathsf{E}(\mathsf{k}_{\mathrm{SKE}}, \tau \| \mathsf{id})$
 7: Send $\mathsf{trans}_1^{(t)} := (\mathtt{addr}, \mathtt{val})$ to the server
 8: return $\sigma^{(t+1)} := (\mathsf{sc}_w, \mathsf{fc}_w)_{w \in \mathcal{W}^{(t+1)}}$
Server:
10: $\mathsf{Index}[\mathtt{addr}] := \mathtt{val}$
11: return $\mathsf{EDB}^{(t+1)} := (\mathsf{Index}, \mathcal{T}, \mathsf{aux})$

Update$(k, \mathsf{del}, (w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$
Client:
 1: if $\mathsf{fc}_w$ is defined then
 2:     $\tau \leftarrow \pi(\mathsf{k}_{\mathrm{RH}}, w \| \mathsf{id})$
 3:     Send $\mathsf{trans}_1^{(t)} := \tau$ to the server
 4: return $\sigma^{(t+1)} := \sigma^{(t)}$
Server:
 6: $\mathcal{T}' \leftarrow \mathsf{AMQ.Insert}(\mathcal{T}, \tau, \mathsf{aux})$
 7: return $\mathsf{EDB}^{(t+1)} := (\mathsf{Index}, \mathcal{T}', \mathsf{aux})$
```

Fig. 2: Setup and Update of our dynamic SSE scheme Laura.

4.2 Our Construction

Let $\pi : \{0,1\}^* \to \{0,1\}^\lambda$ and $g : \{0,1\}^* \to \{0,1\}^\kappa$ be (variable-input-length) PRF families and $h : \{0,1\}^* \to \{0,1\}^\eta$ be a hash function, where λ and η are polynomials in κ. Let $\mathit{\Pi}_{\mathrm{AMQ}} = (\mathsf{AMQ.Gen}, \mathsf{AMQ.Insert}, \mathsf{AMQ.Delete}, \mathsf{AMQ.Lookup})$ be an AMQ data structure. We propose a dynamic SSE scheme Laura = (Setup, Update, Search) from $\mathit{\Pi}_{\mathrm{AMQ}}$, π, g, and h. The pseudo-codes for Laura are given in Figs. 2 and 3, and we provide overviews of each algorithm below.

Setup: Setup(1^κ). The client generates a secret key $k := (\mathsf{k}_{\mathrm{SKE}}, \mathsf{k}_{\mathrm{PRF}}, \mathsf{k}_{\mathrm{RH}})$, where $\mathsf{k}_{\mathrm{SKE}}$ is an SKE secret key and $\mathsf{k}_{\mathrm{PRF}}$ and k_{RH} are PRF keys used to compute ad-

Algorithm: Laura

$\mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$

```
Client:
 1: $\mathsf{K}_q^{(\mathsf{sc}_w)} \leftarrow g(\mathsf{k}_{\mathrm{PRF}}, q \| \mathsf{sc}_q)$
 2: Send $\mathsf{trans}_1^{(t)} := (\mathsf{K}_q^{(\mathsf{sc}_w)}, \mathsf{fc}_q)$ to the server
Server:
 3: for $i = 1$ to $\mathsf{fc}_q$ do
 4:     $\mathtt{addr} \leftarrow h(\mathsf{K}_q^{(\mathsf{sc}_w)}, i)$,   $\mathtt{val} := \mathsf{Index}[\mathtt{addr}]$,   $\mathcal{C}_q^{(t)} \leftarrow \mathtt{val}$
 5:     $\mathsf{Index}[\mathtt{addr}] := \mathrm{NULL}$ // delete old addresses
 6: Send $\mathsf{trans}_2^{(t)} := (\mathcal{C}_q^{(t)}, \mathcal{T}, \mathsf{aux})$ to the client // Send copy of $\mathcal{T}$
Client:
 7: for $\forall \mathsf{c} \in \mathcal{C}_q^{(t)}$ do
 8:     $\tau \| \mathsf{id} \leftarrow \mathsf{D}(\mathsf{k}_{\mathrm{SKE}}, \mathsf{c})$ // the first $\lambda$ MSBs of $\mathtt{val}$ is tag
 9:     if $\mathsf{AMQ.Lookup}(\mathcal{T}, \tau, \mathsf{aux}) = \mathtt{true}$ then // logical deletion of $(q, \mathsf{id})$
10:         $\mathcal{D}_q^{(t)} \leftarrow \tau$
11:     else // search result
12:         $\mathcal{X}_q^{(t)} \leftarrow \mathsf{id}$,   $\mathcal{Y}_q^{(t)} \leftarrow (\tau, \mathsf{id})$
13: $\mathsf{sc}_q := \mathsf{sc}_q + 1$,   $\mathsf{fc}_q := |\mathcal{X}_q^{(t)}|$ // update state
14: $\widehat{\mathsf{K}}_q^{(\mathsf{sc}_q)} \leftarrow g(\mathsf{k}_{\mathrm{PRF}}, q \| \mathsf{sc}_q)$ // generate new keys
15: $\mathsf{ctr} := 1$
16: for $\forall (\tau, \mathsf{id}) \in \mathcal{Y}_q^{(t)}$ do
17:     $\widehat{\mathtt{addr}} \leftarrow h(\widehat{\mathsf{K}}_q^{(\mathsf{sc}_q)}, \mathsf{ctr})$,   $\widehat{\mathtt{val}} \leftarrow \mathsf{E}(\mathsf{k}_{\mathrm{SKE}}, \tau \| \mathsf{id})$
18:     $\mathcal{R}_q^{(t)} \leftarrow (\widehat{\mathtt{addr}}, \widehat{\mathtt{val}})$,   $\mathsf{ctr} := \mathsf{ctr} + 1$
19: Send $\mathsf{trans}_3^{(t)} := (\mathcal{D}_q^{(t)}, \mathcal{R}_q^{(t)})$ to the server
20: return $(\mathcal{X}_q^{(t)}, \sigma^{(t+1)} := (\mathsf{sc}_q, \mathsf{fc}_q)_{q \in \mathcal{W}^{(t+1)}})$
Server:
21: for $\forall (\widehat{\mathtt{addr}}, \widehat{\mathtt{val}}) \in \mathcal{R}_q^{(t)}$ do
22:     $\mathsf{Index}[\widehat{\mathtt{addr}}] := \widehat{\mathtt{val}}$ // set new addresses and value
23: for $\forall \tau \in \mathcal{D}_q^{(t)}$ do
24:     $\mathcal{T}' \leftarrow \mathsf{AMQ.Delete}(\mathcal{T}, \tau, \mathsf{aux})$,   $\mathcal{T} := \mathcal{T}'$
25: return $\mathsf{EDB}^{(t+1)} := (\mathsf{Index}, \mathcal{T}, \mathsf{aux})$
```

Fig. 3: Search of our dynamic SSE scheme Laura.

dresses and tags, respectively. The client initializes two counters fc_w and sc_w, an array Index, and an AMQ data structure $\mathcal{T}$ (along with its auxiliary information aux). The client sets the state information $\sigma^{(0)} := (\mathsf{fc}_w, \mathsf{sc}_w)$, and sends $\mathsf{EDB}^{(0)} := (\mathsf{Index}\mathcal{T}, \mathsf{aux})$ to the server.

Addition: $\mathsf{Update}(k, \mathsf{add}, (w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$. First, the client retrieves the file counter fc_w and the search counter sc_w in $\sigma^{(t)}$ and increments fc_w. The client next derives a PRF key $\mathsf{K}_w^{(\mathsf{sc}_w)}$ from the PRF key $\mathsf{k}_{\mathrm{PRF}}$ using the keyword w to calculate an address $\mathtt{addr}$. Also, the client computes a tag τ, which will be sent

to the server during the deletion operation, of the pair (w, id) from the PRF key k_{RH}, and encrypts $\tau \| \mathsf{id}$ with the SKE secret key $\mathsf{k}_{\mathrm{SKE}}$. The server adds the ciphertext to $\mathsf{Index}[\mathtt{addr}]$ in $\mathsf{EDB}^{(t)}$.
Deletion: $\mathsf{Update}(k, \mathsf{del}, (w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$. The client only computes the tag τ of the pair (w, id) using the PRF key k_{RH} and sends it to the server. The server executes $\mathsf{AMQ.Insert}$ to insert τ into the data structure $\mathcal{T}$ in $\mathsf{EDB}^{(t)}$.

Search: $\mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$. First, the client creates the PRF key $\mathsf{K}_q^{(\mathsf{sc}_w)}$ for the search keyword q and sends it together with fc_q to the server. For every $i = 1, \ldots, \mathsf{fc}_q$, the server computes an address $g(\mathsf{K}_q^{(\mathsf{sc}_w)}, i)$ and adds its stored value $\mathtt{val}$ to the set $\mathcal{C}_q^{(t)}$. The server sends $\mathcal{C}_q^{(t)}$ and a copy of the data structure $\mathcal{T}$ to the client and frees the memory of all the addresses accessed. For every value $\mathtt{val} \in \mathcal{C}_q^{(t)}$, the client checks whether it has been deleted as follows. The client decrypts $\mathtt{val}$ and obtains $\tau \| \mathsf{id}$, and executes $\mathsf{AMQ.Lookup}$ with τ to check whether the pair (w, id) has been logically deleted. If $\mathsf{AMQ.Lookup}$ outputs $\mathtt{false}$, id is added to the search result $\mathcal{X}_q^{(t)}$. Next, the client re-adds the pairs (w, id) except for the deleted ones. The client increments sc_q, and adds the pairs in the same way to the above addition procedure. The server updates $\mathsf{EDB}^{(t)}$ as in the addition procedure and also receives a tag set $\mathcal{D}_q^{(t)}$ of the deleted entry. For every tag $\tau \in \mathcal{D}_q^{(t)}$, the server executes $\mathsf{AMQ.Delete}$ to remove the tags from the data structure $\mathcal{T}$. This re-addition procedure is important to provide forward privacy and reduce the size of EDB and $\mathcal{T}$.

4.3 Security Analysis

Correctness. Before analyzing the security of Laura, we show that it satisfies the correctness. Laura might output wrong search results due to false positives in the underlying AMQ data structure Π_{AMQ}. The correctness error probability depends on the false-positive probability; due to the bounded false-positive probability property, there exists, and we can evaluate an upper bound μ_n of the false-positive probability. Therefore, by setting the parameters of Π_{AMQ} appropriately, one can make the correctness error probability negligible.

Security. To show the security of Laura, we consider a leakage function called *deletion pattern* $\mathsf{DelTime}_q^{(t)}$, which is a set of counters for all deletion operations on w. Namely,

$$\mathsf{DelTime}_q^{(t)} := \left\{ u^{\mathsf{del}} \;\middle|\; \begin{array}{l} \exists \mathsf{id} \text{ s.t. } (u^{\mathsf{add}}, \mathsf{add}, (q, \mathsf{id})) \in \mathcal{Q}^{(t)} \\ \wedge\ (u^{\mathsf{del}}, \mathsf{del}, (q, \mathsf{id})) \in \mathcal{Q}^{(t)} \end{array} \right\},$$

where we assume $u^{\mathsf{add}} < u^{\mathsf{del}}$ without the loss of generality.

Theorem 1. *If Π_{SKE} is* CPA*-secure, Π_{AMQ} is an AMQ data structure, π and g are (variable-input-length) PRF families, and h is a random oracle, the dynamic SSE scheme* $\mathsf{Laura} = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Search})$ *in Figs. 2 and 3 is an $\mathcal{L}$-adaptively* *secure result-hiding scheme that supports forward privacy and Type-*II *backward privacy, with the following leakage function* $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Upd}}, \mathcal{L}_{\mathsf{Srch}})$*:*

$$\mathcal{L}_{\mathsf{Setup}}(1^{\kappa}) = \Lambda, \quad \mathcal{L}_{\mathsf{Upd}}(t, \mathsf{op}, \mathsf{in}) = (t, \mathsf{op}),$$
$$\mathcal{L}_{\mathsf{Srch}}(t, q) = (\mathsf{SP}_q^{(t)}, \mathsf{Update}_q^{(t)}, \mathsf{DelTime}_q^{(t)}),$$

for any t *and any* $q \in \Lambda$*.*

Note that $\mathsf{DelTime}_q^{(t)}$ can be derived from $\mathsf{Update}_q^{(t)}$ and op included in $\mathcal{L}_{\mathsf{Upd}}$: $\mathsf{DelTime}_q^{(t)} := \{u \in \mathsf{Update}_q^{(t)} \mid \mathcal{L}_{\mathsf{Upd}}(u, \mathsf{op}, (q, \mathsf{id})) = (u, \mathsf{del})\}$. Since $\mathsf{Time}_q^{(t)}$ and $\mathsf{DelHist}_q^{(t)}$ imply $\mathsf{Update}_q^{(t)}$, our construction clearly meets both Type-II backward privacy and the result-hiding property.

Proof (Sketch). Due to the page limitation, we give a proof sketch. We will provide the detailed proof in the full version. We prove that the simulator S can simulate the update and search operations only with the leakage functions $\mathcal{L}$.

Addition. With leakage $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{add}, \mathsf{in}) = (t, \mathsf{add})$ for a query $(\mathtt{upd}, \mathsf{add}, \mathsf{in})$, S simulates a transcript $\mathsf{trans}_1^{(t)} := (\mathtt{addr}, \mathsf{c})$. In the real experiment Real, $\mathtt{addr}$ and c are η-bit pseudo-random numbers and ciphertexts of $\tau \| \mathsf{id}$, respectively. If h is a random oracle and Π_{SKE} is CPA-secure, $\mathtt{addr}$ and c are indistinguishable from an η-bit random string r and a ciphertext c' of $0^{\lambda+l}$, except with negligible probability, respectively. Hence, S can set $\mathsf{trans}_1^{(t)} := (r, \mathsf{c}')$.

Deletion. With leakage $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{del}, \mathsf{in}) = (t, \mathsf{del})$ for a query $(\mathtt{upd}, \mathsf{del}, \mathsf{in})$, S simulates a transcript $\mathsf{trans}_1^{(t)} := \tau$. If π is a PRF family, τ is indistinguishable from a λ-bit random string r' except with negligible probability. Therefore, S can set $\mathsf{trans}_1^{(t)} := r'$.

Search. With leakage $\mathcal{L}_{\mathsf{Srch}}(t, q) = (\mathsf{SP}_q^{(t)}, \mathsf{Update}_q^{(t)}, \mathsf{DelTime}_q^{(t)})$ for a query $(\mathtt{srch}, q)$, S simulates transcripts $\mathsf{trans}_1^{(t)} := (\mathsf{K}_q^{(\mathsf{sc}_w)}, \mathsf{fc}_q)$, $\mathsf{trans}_2^{(t)} := (\mathcal{C}_q^{(t)}, \mathcal{T}, \mathsf{aux})$, and $\mathsf{trans}_3^{(t)} := (\mathcal{D}_q^{(t)}, \mathcal{R}_q^{(t)})$. Roughly speaking, due to the security of the underlying PRF g, S can set a κ-bit random string as $\mathsf{K}_q^{(\mathsf{sc}_w)}$. Since fc_q can be derived from $\mathsf{Update}_q^{(t)}$ and $\mathsf{DelTime}_q^{(t)}$, S can simulate $\mathsf{trans}_1^{(t)}$. Since $\mathcal{C}_q^{(t)}$ is a set of all ciphertexts generated during the addition operation for q, S retrieves a ciphertext simulated at every $u \in \mathsf{Update}_q^{(t)} \setminus \mathsf{DelTime}_q^{(t)}$ and sets them as $\mathcal{C}_q^{(t)}$.[4] S easily simulates $\mathcal{T}$ and aux since tags for w, which are entered into $\mathsf{AMQ.Insert}$ and $\mathsf{AMQ.Delete}$, are correctly simulated during the deletion operation. Hence, S can simulate $\mathsf{trans}_2^{(t)}$. The set $\mathcal{D}_q^{(t)}$ of deleted tags can also be simulated as above. $\mathcal{R}_q^{(t)}$ can be simulated as in the case of the addition since each $(\widehat{\mathtt{addr}}, \widehat{\mathtt{val}}) \in \mathcal{R}_q^{(t)}$ is generated in the same manner as the addition operation. Therefore, S can simulate $\mathsf{trans}_3^{(t)}$. $\square$

[4] To be precise, S has to change the way to retrieve ciphertexts depending on $\mathsf{SP}_q^{(t)}$; S first retrieves ciphertexts re-added at the last search for q, i.e., at $t' = \max \mathsf{SP}_q^{(t)}$, and then retrieves ciphertexts simulated from t' to t.

Algorithm: v-Laura

```
Setup($1^\kappa$)
Client:
 1: $\mathsf{k}_{\mathrm{PRF}}, \mathsf{k}_{\mathrm{RH}}, \mathsf{k}_{\mathrm{SKE}} \xleftarrow{\$} \{0,1\}^\kappa$
 2: $\mathsf{fc}_w, \mathsf{sc}_w, \mathbf{F}[], \mathsf{Index}[], \mathsf{Cache}[] := \varepsilon$ // $\varepsilon$ is an empty value
 3: return $\Big(k := (\mathsf{k}_{\mathrm{PRF}}, \mathsf{k}_{\mathrm{RH}}, \mathsf{k}_{\mathrm{SKE}}), \sigma^{(0)} := (\mathsf{sc}_w, \mathsf{fc}_w, \mathbf{F}), \mathsf{EDB}^{(0)} := (\mathsf{Index}, \mathsf{Cache})\Big)$

Update($k$, add, $(w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)}$)
Client:
 1: $\tau \leftarrow \pi(\mathsf{k}_{\mathrm{RH}}, w\|\mathsf{id})$
 2: if $\mathsf{sc}_w$ is undefined then
 3:     $(\mathsf{sc}_w, \mathsf{fc}_w) := (0, 0)$
 4:     $(\mathcal{T}_w, \mathsf{aux}) \leftarrow \mathsf{AMQ.Gen}(\{0,1\}^\lambda, \mathsf{par})$
 5:     $\mathbf{F}[w] := (\mathcal{T}_w, \mathsf{aux})$
 6: $\mathsf{fc}_w := \mathsf{fc}_w + 1$ // increment $\mathsf{fc}_w$
 7: $\mathsf{K}_w^{(\mathsf{sc}_w)} \leftarrow g(\mathsf{k}_{\mathrm{PRF}}, w\|\mathsf{sc}_w)$ // generate the PRF key for address
 8: $\mathsf{c} \leftarrow \mathsf{E}(\mathsf{k}_{\mathrm{SKE}}, \mathsf{id}; \tau)$ // Encryption with nonce
 9: $\mathtt{addr} \leftarrow h(\mathsf{K}_w^{(\mathsf{sc}_w)}, \mathsf{fc}_w)$,    $\mathtt{val} := \tau\|\mathsf{c}$
10: Send $\mathsf{trans}_1^{(t)} := (\mathtt{addr}, \mathtt{val})$ to the server
11: return $\sigma^{(t+1)} := \big((\mathsf{sc}_w, \mathsf{fc}_w)_{w \in \mathcal{W}^{(t+1)}}, \mathbf{F}\big)$
Server:
12: $\mathsf{Index}[\mathtt{addr}] := \mathtt{val}$
13: return $\mathsf{EDB}^{(t+1)} := (\mathsf{Index}, \mathsf{Cache})$

Update($k$, del, $(w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)}$)
Client:
 1: if $\mathsf{fc}_w$ is defined then
 2:     $\tau \leftarrow \pi(\mathsf{k}_{\mathrm{RH}}, w\|\mathsf{id})$
 3:     $(\mathcal{T}_w, \mathsf{aux}) \leftarrow \mathbf{F}[w]$
 4:     $\mathcal{T}'_w \leftarrow \mathsf{AMQ.Insert}(\mathcal{T}_w, \tau, \mathsf{aux})$
 5:     $\mathbf{F}[w] := (\mathcal{T}'_w, \mathsf{aux})$
 6: return $\sigma^{(t+1)} := \big((\mathsf{sc}_w, \mathsf{fc}_w)_{w \in \mathcal{W}^{(t+1)}}, \mathbf{F}\big)$
```

Fig. 4: Setup and Update of our dynamic SSE scheme v-Laura.

5 Extensions

5.1 A Variant of Laura: v-Laura

Although Laura is very efficient with small client storage, there is a trade-off between it and the communication cost, as noted in the footnote in Sec. 4.1. Specifically, the server has to send the AMQ structure together with a search result during the search algorithm (line 6 in Fig. 3). The idea to reduce communication cost is to store the AMQ structure on the client side for each keyword, as in Aura. For clients with ample storage or narrow bandwidth, a more suitable

```
Algorithm: v-Laura
Search$(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$
Client:
 1: $\mathsf{K}_q^{(\mathsf{sc}_w)} \leftarrow g(\mathsf{k}_{\text{PRF}}, q \| \mathsf{sc}_q)$
 2: $\mathsf{tkn}_q \leftarrow g(\mathsf{k}_{\text{PRF}}, q)$
 3: $(\mathcal{T}_q, \mathsf{aux}) := \mathbf{F}[q]$
 4: Send $\mathsf{trans}_1^{(t)} := \left(\mathsf{K}_q^{(\mathsf{sc}_w)}, \mathsf{tkn}_q, \mathsf{fc}_q, (\mathcal{T}_q, \mathsf{aux})\right)$ to the server
Server:
 5: $\mathcal{C}_q^{(t)} := \mathsf{Cache}[\mathsf{tkn}_q]$
 6: for $i = 1$ to $\mathsf{fc}_q$ do
 7:     $\mathtt{addr} \leftarrow h(\mathsf{K}_q^{(\mathsf{sc}_w)}, i)$,   $\mathtt{val} := \mathsf{Index}[\mathtt{addr}]$,   $\mathcal{C}_q^{(t)} \leftarrow \mathtt{val}$
 8:     $\mathsf{Index}[\mathtt{addr}] :=$ NULL // delete old addresses
 9: for $\forall \mathtt{val} \in \mathcal{C}_q^{(t)}$ do
10:     parse $\mathtt{val} = \tau \| \mathsf{c}$ // the first $\lambda$ MSBs of $\mathtt{val}$ is tag(nonce)
11:     if $\mathsf{AMQ.Lookup}(\mathcal{T}_q, \tau, \mathsf{aux}) = \mathtt{true}$ then // logical deletion of $(w, \mathsf{id})$
12:         $\mathcal{C}_q^{(t)} := \mathcal{C}_q^{(t)} \setminus \{\mathtt{val}\}$
13: $\mathsf{Cache}[\mathsf{tkn}_q] := \mathcal{C}_q^{(t)}$
14: Send $\mathsf{trans}_2^{(t)} := \mathcal{C}_q^{(t)}$ to the client
15: return $\mathsf{EDB}^{(t+1)} := (\mathsf{Index}, \mathsf{Cache})$
Client:
 7: for $\forall(\tau, \mathsf{c}) \in \mathcal{C}_q^{(t)}$ do
 8:     $\mathcal{X}_q^{(t)} \leftarrow \mathsf{D}(\mathsf{k}_{\text{SKE}}, \mathsf{c}; \tau)$ // decrypt $\mathsf{c}$ to get search result
 9: $(\mathcal{T}'_q, \mathsf{aux}) \leftarrow \mathsf{AMQ.Gen}(\{0,1\}^\lambda, \mathsf{par})$
10: $\mathsf{fc}_q := 0$,   $\mathsf{sc}_q := \mathsf{sc}_q + 1$,   $\mathbf{F}[q] := (\mathcal{T}'_q, \mathsf{aux})$ // update state
11: return $\left(\mathcal{X}_q^{(t)}, \sigma^{(t+1)} := \left((\mathsf{sc}_q, \mathsf{fc}_q)_{q \in \mathcal{W}^{(t+1)}}, \mathbf{F}\right)\right)$
```

Fig. 5: Search of our dynamic SSE scheme v-Laura.

and efficient variant scheme than Laura, called v-Laura, can be constructed. At first glance, it seems to be the same as Aura, but the following are differences;

1) AMQ is used only as a compression of the deleted tag set without SRE functionality. Therefore, efficient AMQs can be selected, not limited to the bloom filter used for SRE in Aura. The v-Laura also achieves efficient search by eliminating SRE processing, which is dominant in Aura searches (see Sec. 5).
2) The server removes the deleted entries using AMQ structure while the client decrypts the search results to achieve result-hiding, similar to Laura.
3) The v-Laura can compress the size of val in EDB with the idea of using τ as a nonce in encryption. In some block cipher modes of CPA-secure Π_{SKE}, the nonce is used for security and is stored with the ciphertext. Since τ plays the role of nonce, it can compress the size of the original nonce.

The pseudo-codes for $\mathsf{v\text{-}Laura}$ are given in Figs. 4 and 5, and we provide overviews of each algorithm below. However, we omit the same part of Laura.

Setup: $\mathsf{Setup}(1^\kappa)$. The client generates a secret key $k := (\mathsf{k}_{\mathrm{SKE}}, \mathsf{k}_{\mathrm{PRF}}, \mathsf{k}_{\mathrm{RH}})$. The client initializes two counters fc_w and sc_w, and three array Index and Cache and $\mathbf{F}$. The client sets the state information $\sigma^{(0)} := (\mathsf{fc}_w, \mathsf{sc}_w, \mathbf{F})$, and sends $\mathsf{EDB}^{(0)} := (\mathsf{IndexCache})$ to the server.

Addition: $\mathsf{Update}(k, \mathsf{add}, (w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$. First, the client calculates an address $\mathtt{addr}$ and tag τ, like Laura. Also, the client encrypts id using τ as nonce (i.e., $\mathsf{c} \leftarrow \mathsf{E}(\mathsf{k}_{\mathrm{SKE}}, \mathsf{id}; \tau)$) and sends $\mathtt{addr}$ and $\mathtt{val} := \tau \| \mathsf{c}$ to the server. The server adds $\mathtt{val}$ to $\mathsf{Index}[\mathtt{addr}]$ in $\mathsf{EDB}^{(t)}$.
Deletion: $\mathsf{Update}(k, \mathsf{del}, (w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$. The client only computes the tag τ and executes $\mathsf{AMQ.Insert}$ to insert τ into the data structure $\mathcal{T}_w$ for w in $\sigma^{(t)}$.

Search: $\mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$. First, the client creates $\mathsf{K}_q^{(\mathsf{sc}_w)}$ and tkn_q with the PRF key $\mathsf{k}_{\mathrm{PRF}}$ and sends them together with fc_q and $\mathcal{T}_q$ to the server. The server gets $\mathsf{Cache}[\mathsf{tkn}_q]$ as a set $\mathcal{C}_q^{(t)}$. For every $i = 1, \ldots, \mathsf{fc}_q$, the server computes an address $g(\mathsf{K}_q^{(\mathsf{sc}_w)}, i)$ and adds its stored value $\mathtt{val}$ to the set $\mathcal{C}_q^{(t)}$. For every $\mathtt{val} \in \mathcal{C}_q^{(t)}$, the server parse $\mathtt{val} := \tau \| \mathsf{c}$ and executes $\mathsf{AMQ.Lookup}$ with τ and $\mathcal{T}_q$ to check whether the pair (q, id) has been logically deleted. If $\mathsf{AMQ.Lookup}$ outputs $\mathtt{true}$, $\mathtt{val}$ is removed from $\mathcal{C}_q^{(t)}$. Next, the server sets $\mathcal{C}_q^{(t)}$ to $\mathsf{Cache}[\mathsf{tkn}_q]$ and updates $\mathsf{EDB}^{(t)}$, and sends $\mathcal{C}_q^{(t)}$ to the client. For every value $\mathtt{val} \in \mathcal{C}_q^{(t)}$, the client decrypts $\mathtt{val}$ to obtain id and adds it to the search result $\mathcal{X}_q^{(t)}$. Finally, the client initializes $\mathcal{T}_q$ and fc_q and increments sc_q.

$\mathsf{v\text{-}Laura}$ also satisfies Theorem 1. The proof is shown in full version.

5.2 A Strongly Secure variant of Laura: s-Laura

As explained in the introduction, Aura implicitly requires every pair of (w, id) to be added at most only once; it does not allow the client to re-add previously deleted pairs. Indeed, Laura and $\mathsf{v\text{-}Laura}$ work well under the same assumption. In other words, if the client wants to add and delete a pair (w, id) multiple times, those schemes are no longer Type-II backward private. This limitation stems from the fact that the corresponding tag of the pair (w, id) is generated deterministically in those schemes. The extended scheme $\mathsf{s\text{-}Laura}$, which stands for strongly-secure Laura, allows to run Update of pair (w, id) any number of times. The basic idea of $\mathsf{s\text{-}Laura}$ is that the deletion tag of the pair (w, id) changes with each deletion. The client holds extra information dc_w which increments for each deletion regarding w. When pair (w, id) is deleted, a delete tag τ_{dc_w} is generated from τ and dc_w. The client then computes tags $\tau_1, \ldots, \tau_{\mathsf{dc}_w}$ from τ and dc_w, and executes $\mathsf{AMQ.Lookup}$ with τ_i for every $i \in [\mathsf{dc}_w]$ to check whether the pair (w, id) has been logically deleted. If $\mathsf{AMQ.Lookup}$ outputs $\mathtt{false}$ for all tags, id is added to the search result $\mathcal{X}_q^{(t)}$. However, the search time of $\mathsf{s\text{-}Laura}$ increases linearly with the number of deletions, as shown in Table. 1. Hence, $\mathsf{s\text{-}Laura}$ has not been evaluated for implementation in Sec. 6. Efficient construction is a future work.

We give the pseudo-codes for s-Laura in Appendix A, and provide overviews of each algorithm below.

Setup: $\mathsf{Setup}(1^\kappa)$. The client generates a secret key $k := (\mathsf{k}_{\mathrm{SKE}}, \mathsf{k}_{\mathrm{PRF}}, \mathsf{k}_{\mathrm{RH}})$, where $\mathsf{k}_{\mathrm{SKE}}$ is an SKE secret key and $\mathsf{k}_{\mathrm{PRF}}$ and k_{RH} are PRF keys used to compute addresses and tags, respectively. The client initializes three counters fc_w, sc_w, and dc_w, an array Index, and an AMQ data structure $\mathcal{T}$ (along with its auxiliary information aux). The client sets the state information $\sigma^{(0)} := (\mathsf{fc}_w, \mathsf{sc}_w, \mathsf{dc}_w)$, and sends $\mathsf{EDB}^{(0)} := (\mathsf{Index}\mathcal{T}, \mathsf{aux})$ to the server.

Addition: $\mathsf{Update}(k, \mathsf{add}, (w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$. First, the client retrieves the file counter fc_w and the search counter sc_w in $\sigma^{(t)}$ and increments fc_w. The client next derives a PRF key $\mathsf{K}_{w,0}^{(\mathsf{sc}_w)}$ from the PRF key $\mathsf{k}_{\mathrm{PRF}}$ using the keyword w to calculate an address $\mathtt{addr}$. Also, the client computes a *persistent tag* τ, which will be used to derive an *ephemeral tag* τ_i during the deletion operation, of the pair (w, id) from the PRF key k_{RH}, and encrypts $\tau \| \mathsf{id}$ with the SKE secret key $\mathsf{k}_{\mathrm{SKE}}$. The server adds the ciphertext to $\mathsf{Index}[\mathtt{addr}]$ in $\mathsf{EDB}^{(t)}$.

Deletion: $\mathsf{Update}(k, \mathsf{del}, (w, \mathsf{id}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$. First, the client retrieves the deletion counter dc_w and increments it. The client computes the persistent tag τ as in the addition operation. Then, the client derives a key $\mathsf{K}_{w,1}^{(\mathsf{sc}_w)}$ from the PRF key $\mathsf{k}_{\mathrm{PRF}}$ using the keyword w and generates an ephemeral tag τ_{dc_w} from the derived key $\mathsf{K}_{w,1}^{(\mathsf{sc}_w)}$, the persistent tag τ, and the counter dc_w. The server executes $\mathsf{AMQ.Insert}$ to insert τ into the data structure $\mathcal{T}$ in $\mathsf{EDB}^{(t)}$.

Search: $\mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$. First, the client creates the PRF key $\mathsf{K}_{q,0}^{(\mathsf{sc}_w)}$ for the search keyword q and sends it together with fc_q to the server. For every $i = 1, \ldots, \mathsf{fc}_q$, the server computes an address $g(\mathsf{K}_{w,0}^{(\mathsf{sc}_w)}, i)$ and adds its stored value $\mathtt{val}$ to the set $\mathcal{C}_q^{(t)}$. The server sends $\mathcal{C}_q^{(t)}$ and a copy of the data structure $\mathcal{T}$ to the client and frees the memory of all the addresses accessed. For every value $\mathtt{val} \in \mathcal{C}_q^{(t)}$, the client checks whether it has been deleted as follows. The client decrypts $\mathtt{val}$ and obtains $\tau \| \mathsf{id}$. The client then computes ephemeral tags $\tau_1, \ldots, \tau_{\mathsf{dc}_q}$ from τ and dc_q, and executes $\mathsf{AMQ.Lookup}$ with τ_i for every $i \in [\mathsf{dc}_q]$ to check whether the pair (w, id) has been logically deleted. If $\mathsf{AMQ.Lookup}$ outputs $\mathtt{false}$ for all ephemeral tags, id is added to the search result $\mathcal{X}_q^{(t)}$. Next, the client re-adds the pairs (w, id) except for the deleted ones. The client increments sc_q, and adds the pairs in the same way to the above addition procedure. The server updates $\mathsf{EDB}^{(t)}$ as in the addition procedure and also receives a set $\mathcal{D}_q^{(t)}$ of the ephemeral tags of the deleted entry. For every ephemeral tag $\tau_i \in \mathcal{D}_q^{(t)}$, the server executes $\mathsf{AMQ.Delete}$ to remove the tags from the data structure $\mathcal{T}$. This re-addition procedure is important to provide forward privacy and reduce the size of $\mathcal{T}$.

s-Laura also satisfies Theorem 1. The proof is shown in full version.

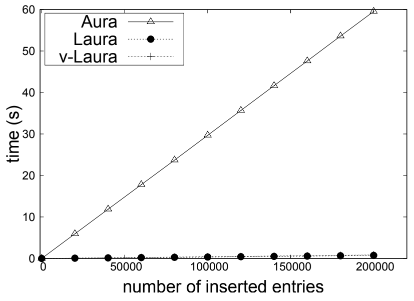

Fig. 6: Addition cost.

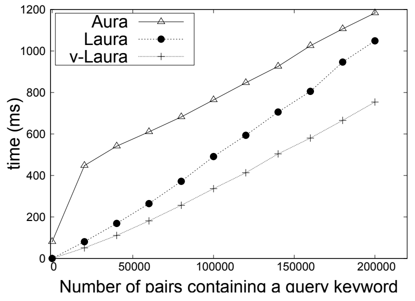

Fig. 7: Search cost without deletion.

6 Experiments

Implementation. We implemented the proposed protocols Laura and v-Laura in C++ and evaluated their performance comparatively.[5] We compare them with Aura [25] implemented in C++ [1] for each protocol. For instances and technical details of Aura, please refer to [25, 1]. These experiments were done in an Ubuntu 18.04 LTS server with 756GB RAM, using Docker (version 24.0.4) [3]. We used AES-GCM for the instantiation of SKE Π_{SKE}. The PRFs π, g, and the random oracle h are realized with AES-GCM and GMAC, respectively. They are implemented using the EVP functions API on the open SSL library (version 3.0.2 15 Mar 2022), and AES-GCM is accelerated by the Intel AES-NI instruction set. For the instance of the AMQ data structure of Laura and v-Laura, we choose the cuckoo filter [15] implemented in [2].

The sizes of keys and outputs of AES and PRF are 128 bits, respectively. The identifier id and each counter (i.e. $\mathsf{fc}_w, \mathsf{sc}_w$) are 32-bit integers. For experiments on search, we measure the time it takes the server to get all the decrypted identifiers in the search results. Note that both the client and server run locally and communication costs are not taken into account.

Parameter Setting. Throughout the experiments, we set the false-positive probability $p = 10^{-4}$, which was also considered practically acceptable in the Aura paper [25]. To ensure that false-positive probability, we need to set the maximum number d_w of elements inserted into the AMQ data structure in Laura and v-Laura (resp., the Bloom filter in Aura) at the beginning of the protocol. To be precise, Aura and v-Laura prepares a filter per keyword, while Laura employ only one AMQ structure for the whole system. Therefore, unless otherwise stated, we set $d_w = 1{,}000$ for Aura and v-Laura and $d_\Lambda = 10{,}000{,}000$ for Laura, where d_w and $d_\Lambda = \sum_{w \in \Lambda} d_w$.

Addition Cost. We give the addition costs of Aura, Laura, and v-Laura in Fig. 6. This results surprisingly show a marked performance difference between

[5] We did not implement *sOurs* since we want to compare dynamic SSE schemes with the same security level. Note that s-Laura is secure even if deleted entries are re-added.

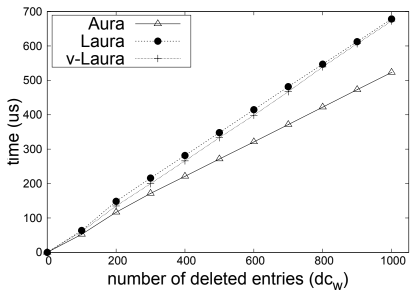

Fig. 8: Deletion cost.

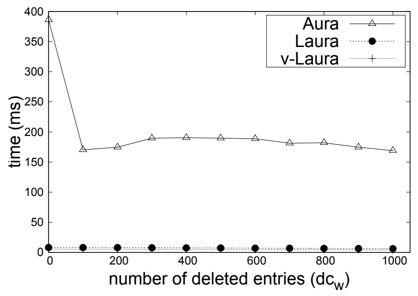

Fig. 9: Search cost with deletion.

ours and Aura. Specifically, Laura and v-Laura takes less than 1.0 s to add 200,000 keyword-identifier pairs, whereas Aura takes 59.5 s. This is due to the concrete construction of the underlying SRE scheme, which requires many resources for the addition.

Search Cost without Deletion. Fig. 7 compares the search costs of Aura, Laura, and v-Laura when no entries on w have been deleted. The search costs of the three schemes increase linearly with the number of pairs. When the search results is 200,000 pairs, Laura, v-Laura, and Aura take 1.05 s, 0.75 s and 1.18 s respectively.

Deletion Cost. As can be seen in Fig. 8, the deletion costs for Aura, Laura, and v-Laura are remarkably fast since the deletion procedures of these schemes only require the calculation of the tag corresponding to the pair to be deleted and the insertion to the filter. Specifically, for 1,000 deleted entries, Laura, v-Laura and Aura take 0.68 ms, 0.67 ms and 0.52 ms respectively. The Laura and v-Laura are slightly slower since the cuckoo filter [15] has the property that as more items are inserted to the filter, the frequency of kicked out an item in the insertion also increases.

Search Cost with Deletion. We show the effect of deletion on search costs in Fig. 9. After adding 2,000 pairs of (w, id), we delete pairs and then search for w. Fig. 9 shows the search time with the range of the number of the deleted pairs from 0 to 1,000. The Laura and v-Laura are remarkably faster than Aura. Specifically, when deleting 1,000 entries (i.e., 1,000 results of 2,000 entries), Laura, v-Laura and Aura take 0.61 ms, 0.41 ms and 169.0 ms respectively. Compared Aura with v-Laura, it is clear that the computational complexity of SRE is dominant. More interestingly, Aura takes longer when no deletion occurred due to the underlying SRE construction.

Acknowledgment This work was supported by JSPS KAKENHI Grant Numbers JP21H03395, JP21H03441, JP22H03590, JP23H00468, JP23H00479, JP23K17455, JST CREST JPMJCR23M2, and MEXT Leading Initiative for Excellent Young Researchers.

References

1. Aura. `https://github.com/MonashCybersecurityLab/Aura`
2. Cuckoo filter. `https://github.com/efficient/cuckoofilter/tree/master`
3. Docker. `https://www.docker.com/`
4. Bender, M.A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B.C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R.P., Zadok, E.: Don't thrash: How to cache your hash on flash. Proc. VLDB Endow. **5**(11), 1627–1637 (2012)
5. Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. In: NDSS 2020. The Internet Society (2020)
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
7. Bost, R.: $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In: Proc. of ACM CCS 2016. pp. 1143–1154. ACM (2016)
8. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proc. of ACM CCS 2017. pp. 1465–1482. ACM (2017)
9. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proc. of ACM CCS 2015. pp. 668–679. ACM (2015)
10. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. In: Proc. of NDSS 2014. The Internet Society (2014)
11. Chamani, J.G., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: Proc. of ACM CCS 2018. pp. 1038–1055. ACM (2018)
12. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proc. of ACM CCS 2006. pp. 79–88. ACM (2006)
13. Demertzis, I., Chamani, J.G., Papadopoulos, D., Papamanthou, C.: Dynamic searchable encryption with small client storage. In: Proc. of NDSS 2020. The Internet Society (2020)
14. Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. PoPETs **2018**(1), 5–20 (2018)
15. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: Practically better than bloom. In: Proc. of CoNEXT 2014. pp. 75–88 (2014)
16. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: ACM SIGSAC Conference on Computer and Communications Security, CCS 2014. pp. 310–320. ACM, New York, NY, USA (2014)
17. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: Proc. of NDSS 2012. The Internet Society (2012)
18. Kamara, S., Moataz, T.: Computationally volume-hiding structured encryption. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2019. pp. 183–213. Springer International Publishing, Cham (2019)
19. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Proc. of FC 2013. pp. 258–274. Springer (2013)
20. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proc. of ACM CCS 2012. pp. 965–976. ACM (2012)
21. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. CRC Press (2014)

22. Miers, I., Mohassel, P.: IO-DSSE: scaling dynamic searchable encryption to millions of indexes by improving locality. In: Network and Distributed System Security Symposium, NDSS 2017 (2017)
23. Naveed, M., Prabhakaran, M., Gunter, C.: Dynamic searchable encryption via blind storage. In: IEEE Symposium on Security and Privacy, S&P 2014. pp. 639–654 (May 2014)
24. Song, D.X., Wagner, D.A., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE S&P 2000. pp. 44–55. IEEE (2000)
25. Sun, S., Steinfeld, R., Lai, S., Yuan, X., Sakzad, A., Liu, J.K., Nepal, S., Gu, D.: Practical non-interactive searchable encryption with forward and backward privacy. In: Proc. of NDSS 2021. The Internet Society (2021)
26. Wang, J., Chow, S.S.M.: Omnes pro uno: Practical multi-writer encrypted database. In: 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022. pp. 2371–2388. USENIX Association (2022)
27. Watanabe, Y., Ohara, K., Iwamoto, M., Ohta, K.: Efficient dynamic searchable encryption with forward privacy under the decent leakage. In: Proc. of ACM CODASPY 2022. pp. 312–323. ACM (2022)
28. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: Proc. of USENIX Security 2016. pp. 707–720. USENIX Association (2016)

A Formal Description of s-Laura

We give the concrete procedures of s-Laura in Figs 10 and 11.

Algorithm: s-Laura

```
Setup(1^κ)
Client:
 1: $\mathsf{k}_{\text{PRF}}, \mathsf{k}_{\text{RH}}, \mathsf{k}_{\text{SKE}} \xleftarrow{\$} \{0,1\}^\kappa$
 2: $(\mathcal{T}, \mathsf{aux}) \leftarrow \mathsf{AMQ.Gen}(\{0,1\}^\lambda, \mathsf{par})$
 3: $\mathsf{fc}_w, \mathsf{sc}_w, \mathsf{dc}_w, \mathsf{Index}[] \coloneqq \varepsilon$ // $\varepsilon$ is an empty value
 4: return $\Big(k \coloneqq (\mathsf{k}_{\text{PRF}}, \mathsf{k}_{\text{RH}}, \mathsf{k}_{\text{SKE}}), \sigma^{(0)} \coloneqq (\mathsf{sc}_w, \mathsf{fc}_w, \mathsf{dc}_w), \mathsf{EDB}^{(0)} \coloneqq (\mathsf{Index}, \mathcal{T}, \mathsf{aux})\Big)$

Update(k, add, (w, id), σ^(t); EDB^(t))
Client:
 1: $\tau \leftarrow \pi(\mathsf{k}_{\text{RH}}, w\|\mathsf{id})$
 2: if $\mathsf{sc}_w$ is undefined then
 3:     $(\mathsf{sc}_w, \mathsf{fc}_w, \mathsf{dc}_w) \coloneqq (0, 0, 0)$
 4: $\mathsf{fc}_w \coloneqq \mathsf{fc}_w + 1$ // increment $\mathsf{fc}_w$
 5: $\mathsf{K}_{w,0}^{(\mathsf{sc}_w)} \coloneqq g(\mathsf{k}_{\text{PRF}}, w\|\mathsf{sc}_w\|0)$ // generate the PRF key for address
 6: $\mathtt{addr} \leftarrow h(\mathsf{K}_{w,0}^{(\mathsf{sc}_w)}, \mathsf{fc}_w)$
 7: $\mathtt{val} \leftarrow \mathsf{E}(\mathsf{k}_{\text{SKE}}, \tau\|\mathsf{id})$
 8: Send $\mathsf{trans}_1^{(t)} \coloneqq (\mathtt{addr}, \mathtt{val})$ to the server
 9: return $\sigma^{(t+1)} \coloneqq (\mathsf{sc}_w, \mathsf{fc}_w, \mathsf{dc}_w)_{w \in \mathcal{W}^{(t+1)}}$
Server:
10: $\mathsf{Index}[\mathtt{addr}] \coloneqq \mathtt{val}$
11: return $\mathsf{EDB}^{(t+1)} \coloneqq (\mathsf{Index}, \mathcal{T}, \mathsf{aux})$

Update(k, del, (w, id), σ^(t); EDB^(t))
Client:
 1: if $\mathsf{dc}_w$ is defined then
 2:     $\mathsf{dc}_w \coloneqq \mathsf{dc}_w + 1$
 3:     $\tau \leftarrow \pi(\mathsf{k}_{\text{RH}}, w\|\mathsf{id})$
 4:     $\mathsf{K}_{w,1}^{(\mathsf{sc}_w)} \coloneqq g(\mathsf{k}_{\text{PRF}}, w\|\mathsf{sc}_w\|1)$
 5:     $\tau_{\mathsf{dc}_w} \leftarrow \pi(\mathsf{K}_{w,1}^{(\mathsf{sc}_w)}, \tau\|\mathsf{dc}_w)$
 6:     Send $\mathsf{trans}_1^{(t)} \coloneqq \tau_{\mathsf{dc}_w}$ to the server
 7: return $\sigma^{(t+1)} \coloneqq (\mathsf{sc}_w, \mathsf{fc}_w, \mathsf{dc}_w)_{w \in \mathcal{W}^{(t+1)}}$
Server:
 8: $\mathcal{T}' \leftarrow \mathsf{AMQ.Insert}(\mathcal{T}, \tau_{\mathsf{dc}_w}, \mathsf{aux})$
 9: return $\mathsf{EDB}^{(t+1)} \coloneqq (\mathsf{Index}, \mathcal{T}', \mathsf{aux})$
```

Fig. 10: Setup and Update of our dynamic SSE scheme s-Laura.

Algorithm: s-Laura

```
Search$(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$
Client:
 1: $\mathsf{K}_{q,0}^{(\mathsf{sc}_w)} := g(\mathsf{k}_{\mathrm{PRF}}, q \| \mathsf{sc}_w \| 0)$
 2: Send $\mathsf{trans}_1^{(t)} := (\mathsf{K}_{q,0}^{(\mathsf{sc}_w)}, \mathsf{fc}_q)$ to the server
Server:
 3: for $i = 1$ to $\mathsf{fc}_q$ do
 4:     $\mathtt{addr} \leftarrow h(\mathsf{K}_{q,0}^{(\mathsf{sc}_w)}, i)$,    $\mathcal{C}_q^{(t)} \leftarrow \mathsf{Index}[\mathtt{addr}]$
 5:     $\mathsf{Index}[\mathtt{addr}] :=$ NULL // delete old addresses
 6: Send $\mathsf{trans}_2^{(t)} := (\mathcal{C}_q^{(t)}, \mathcal{T}, \mathsf{aux})$ to the client // Send copy of $\mathcal{T}$
Client:
 7: $\mathsf{K}_{q,1}^{(\mathsf{sc}_w)} := g(\mathsf{k}_{\mathrm{PRF}}, q \| \mathsf{sc}_q \| 1)$
 8: for $\forall \mathsf{c} \in \mathcal{C}_q^{(t)}$ do // define Loop1 for Jump
 9:     $\tau \| \mathsf{id} \leftarrow \mathsf{D}(\mathsf{k}_{\mathrm{SKE}}, \mathsf{c})$ // the first $\lambda$ MSBs of $\mathtt{val}$ is tag
10:     for $i = 1$ to $\mathsf{dc}_q$ do
11:         $\tau_i \leftarrow \pi(\mathsf{K}_{q,1}^{(\mathsf{sc}_w)}, \tau \| i)$
12:         if $\mathsf{AMQ.Lookup}(\mathcal{T}, \tau_i, \mathsf{aux}) = \mathtt{true}$ then
13:             $\mathcal{D}_q^{(t)} \leftarrow \tau_i$
14:             Jump Loop1 and next element
15:     $\mathcal{X}_q^{(t)} \leftarrow \mathsf{id}$,    $\mathcal{Y}_q^{(t)} \leftarrow (\mathsf{id}, \tau)$
16: $\mathsf{sc}_q := \mathsf{sc}_q + 1$,    $\mathsf{fc}_q := |\mathcal{X}_q^{(t)}|$,    $\mathsf{dc}_q := 0$ // update state
17: $\widehat{\mathsf{K}}_{q,0}^{(\mathsf{sc}_{q,0})} := g(\mathsf{k}_{\mathrm{PRF}}, q \| \mathsf{sc}_q \| 0)$ // generate new keys
18: $\mathsf{ctr} := 1$
19: for $\forall (\tau, \mathsf{id}) \in \mathcal{Y}_q^{(t)}$ do
20:     $\mathcal{R}_q^{(t)} \leftarrow (h(\widehat{\mathsf{K}}_{q,0}^{(\mathsf{sc}_{q,0})}, \mathsf{ctr}), \mathsf{E}(\mathsf{k}_{\mathrm{SKE}}, \tau \| \mathsf{id}))$ // new $(\widehat{\mathtt{addr}}, \widehat{\mathtt{val}})$ pair
21:     $\mathsf{ctr} := \mathsf{ctr} + 1$
22: Send $\mathsf{trans}_3^{(t)} := (\mathcal{D}_w^{(t)}, \mathcal{R}_q^{(t)})$ to the server
23: return $(\mathcal{X}_q^{(t)}, \sigma^{(t+1)} := (\mathsf{sc}_w, \mathsf{fc}_q, \mathsf{dc}_q)_{q \in \mathcal{W}^{(t+1)}})$
Server:
24: for $\forall (\widehat{\mathtt{addr}}, \widehat{\mathtt{val}}) \in \mathcal{R}_q^{(t)}$ do
25:     $\mathsf{Index}[\widehat{\mathtt{addr}}] := \widehat{\mathtt{val}}$ // set new addresses and value
26: for $\forall \tau_i \in \mathcal{D}_q^{(t)}$ do
27:     $\mathcal{T}' \leftarrow \mathsf{AMQ.Delete}(\mathcal{T}, \tau_i, \mathsf{aux})$,    $\mathcal{T} := \mathcal{T}'$
28: return $\mathsf{EDB}^{(t+1)} := (\mathsf{Index}, \mathcal{T}, \mathsf{aux})$
```

Fig. 11: Search of our dynamic SSE scheme s-Laura.