How to Make a Secure Index for Searchable Symmetric Encryption, Revisited

Yohei Watanabe[1,2], Takeshi Nakai[1], Kazuma Ohara[3], Takuya Nojima[1], Yexuan Liu[*,4], Mitsugu Iwamoto[1], and Kazuo Ohta[1,3]

[1] The University of Electro-Communications, Tokyo, Japan.
{watanabe, t-nakai, mitsugu, kazuo.ohta}@uec.ac.jp

[2] National Institute of Information and Communications Technology (NICT), Tokyo, Japan.

[3] National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan.
ohara.kazuma@aist.go.jp

[4] Yokohama National University, Yokohama, Japan.
liu-yexuan-yb@ynu.jp

July 13, 2021

Abstract

Searchable symmetric encryption (SSE) enables clients to search encrypted data. Curtmola et al. (ACM CCS 2006) formalized a model and security notions of SSE and proposed two concrete constructions called SSE-1 and SSE-2. After the seminal work by Curtmola et al., SSE becomes an active area of encrypted search.

In this paper, we focus on two unnoticed problems in the seminal paper by Curtmola et al. First, we show that SSE-2 does not appropriately implement Curtmola et al.'s construction idea for dummy addition. We refine SSE-2's (and its variants') dummy-adding procedure to keep the number of dummies sufficiently many but as small as possible. We then show how to extend it to the dynamic setting while keeping the dummy-adding procedure work well and implement our scheme to show its practical efficiency. Second, we point out that the SSE-1 can cause a search error when a searched keyword is not contained in any document file stored at a server and show how to fix it.

[*]This work was done while the author was an undergraduate student at The University of Electro-Communications.

Contents

1 Introduction

1.1 Backgrounds

Many services use extensive data on service users and compile databases of the data to accelerate search functions. With the development of the information society, databases increasingly contain sensitive/personal information. One of the important issues to make our information society safer is protecting database privacy while keeping search function efficient. *Searchable symmetric encryption* (SSE), introduced by Song et al. [1], is a cryptographic solution to the challenge and provides a way to efficiently search a large database (e.g., cloud storage) for *encrypted* data. Curtmola et al. [2, 3] first provided a systematic formalization of SSE, and many works (e.g., [4–15]) followed their seminal work.

There are two major and fundamental index data structures; the *forward index*, which stores a mapping from documents to words, and the *inverted index*, which stores a mapping from words to documents. Due to its structure, the latter is usually used for keyword searches; one computes the mapping with a keyword to be searched and obtains information on documents that contain the keyword. Therefore, it ensures efficient search cost since the size often depends on the number of such documents, not all documents. Since the original goal of SSE is to provide efficient keyword search while allowing leakage of inconsequential information, most previous works focused on the inverted-index-based approach. On the other hand, one can use the forward index to search for keywords as well, though it impairs the search efficiency; to search a keyword, one computes the mapping for each document and extracts information on the keyword, and hence, its total size is proportional to the number of documents stored in the index. There are a few works [2, 3, 16, 17] on the forward-index-based approach in contrast to the inverted index[1] since the asymptotic search efficiency is obviously less efficient than the inverted-index-based ones. Hence, the practical efficiency of forward-index-based SSE schemes is still unclear.

In the seminal work, Curtmola et al. [2, 3] showed two concrete SSE schemes via the respective approach. They showed a trade-off on search efficiency and security level between those approaches. The inverted-index-based scheme, called SSE-1, provides a more efficient search procedure than the forward-index-based scheme called SSE-2. On the other hand, SSE-2 can meet adaptive security, whereas SSE-1 only satisfies a weaker notion, called non-adaptive security.[2] Both schemes require dummies to mask the indexes to hide the number of keywords that appear in each document. This paper focuses on how to handle dummies in SSE-1 and SSE-2, respectively.

1.2 Our Contributions

In this paper, we reveal two problems in the seminal paper by Curtmola et al. [2, 3] that have been overlooked thus far.

The original construction idea is not appropriately reflected in SSE-2. We show that SSE-2 and its variants [8, 17] did not implement Curtmola et al.'s construction idea. Specifically, Curtmola et al. described that sufficiently many dummies should be added for each document file, not for each keyword, when creating an encrypted database (or a *secure index*). The idea is indeed compatible with the forward index due to the mapping from documents to words; when creating the secure index, for each document, one computes the mapping and appends sufficiently-many dummies to its output. However, as already pointed out in [8], the original SSE-2 scheme is flawed

[1]Several works [8, 9, 18, 19] implicitly dealt with the approach.

[2]The follow-up works (e.g., [13, 20–22]) showed that the inverted-index-based approach could achieve adaptive security.

in its dummy-adding procedure. Note that the flaw does not stem from the construction idea. Kurosawa and Ohtaki [8] and Hayasaka et al. [17] showed variants of SSE-2 that succeeded in eliminating the bugs. However, their fixes violate Curtmola et al.'s construction idea since they added a lot of dummies for each keyword, not for each document.

Based on the above observation, in Section 3, we revisit those schemes as follows. We show that the above fixes cause the unnecessarily-large secure-index size and propose a reasonable and plausible construction approach that appropriately implements Curtmola et al.'s construction idea. Consequently, we refine SSE-2's dummy-adding procedure to keep the number of dummies sufficiently many but as small as possible. Furthermore, we extend our SSE scheme in the dynamic setting and implement it to unveil the advantages and potential practicality of forward-index-based schemes such as SSE-2. Our dynamic SSE scheme in Section 4 is, thanks to the forward-index-based approach, simple and does not require any state information, which is secret information that might be updated on every update/search operation. Although the forward-index-based approach compounds the search cost in an asymptotic sense, the implementation of our dynamic SSE scheme in Section 5 shows its practical efficiency; the file addition and deletion procedures for a single file take roughly 150μs and 3.8μs, respectively, and the search procedure requires roughly 0.7s when 200,000 files are registered.

SSE-1 does not satisfy the search correctness. We point out that when no document files stored on the server contain a searched keyword, SSE-1 can cause a search error and hence does not meet the search correctness. This error is due to the careless handling of dummies for keywords that do *not* appear in any stored file. Furthermore, we also point out that the original SSE-1 does not work for the large keyword universe (called a dictionary) even if we ignore the above error. In Section 6, we show how to fix the error and modify the scheme to handle a large dictionary.[3]

2 Preliminaries

2.1 Notations

For any positive integer $n \in \mathbb{N}$, $\{1, \ldots, n\}$ is denoted by $[n]$. For a finite set $\mathcal{X}$, we denote by $x \xleftarrow{\$} \mathcal{X}$ and $\mathcal{X} \leftarrow x$ the processes of sampling a value x from $\mathcal{X}$ uniformly at random and adding x to $\mathcal{X}$, respectively, and we use $|\mathcal{X}|$ to represent the cardinality of $\mathcal{X}$. Concatenation and an empty string are denoted by $\|$ and ε, respectively. For algorithm description, all strings, sets, and arrays are initially set to empty ones. Throughout the paper, we denote by κ a security parameter and 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. We denote by $\mathsf{A}^{\mathsf{O}(\cdot)}(\mathsf{in})$ A allowed access to an oracle O. In this paper, we consider two-party interactive algorithms between a client and a server. $\langle \mathsf{out}_{\mathrm{C}}, \mathsf{out}_{\mathrm{S}} \rangle \leftarrow \langle \mathsf{A}_{\mathrm{C}}(\mathsf{in}_{\mathrm{C}}), \mathsf{A}_{\mathrm{S}}(\mathsf{in}_{\mathrm{S}}) \rangle$ means that the client and server run A_{C} and A_{S} with each input in_{C} and in_{S}, respectively, and get each output $\mathsf{out}_{\mathrm{C}}$ and $\mathsf{out}_{\mathrm{S}}$, respectively. For simplicity, we describe the above as $(\mathsf{out}_{\mathrm{C}}; \mathsf{out}_{\mathrm{S}}) \leftarrow \mathsf{A}(\mathsf{in}_{\mathrm{C}}; \mathsf{in}_{\mathrm{S}})$. As necessary we explicitly describe the transcript trans as $\langle (\mathsf{out}_{\mathrm{C}}; \mathsf{out}_{\mathrm{S}}), \mathsf{trans} \rangle \leftarrow \mathsf{A}(\mathsf{in}_{\mathrm{C}}; \mathsf{in}_{\mathrm{S}})$. We say a function $\mathsf{negl}(\cdot)$ is negligible if for any polynomial $\mathsf{poly}(\cdot)$, there exists some constant $\kappa_0 \in \mathbb{N}$ such that $\mathsf{negl}(\kappa) < 1/\mathsf{poly}(\kappa)$ for all $\kappa \geq \kappa_0$.

[3]We do not consider a dynamic version and implementations of SSE-1 (more broadly, inverted-index-based schemes) since they are well analyzed in previous works such as [10].

Experiment: $\mathsf{Exp}^{\mathsf{PCPA}}_{\Pi_{\text{SKE}},\mathsf{A}}(\kappa)$

$K \xleftarrow{\$} \mathsf{G}(\kappa)$
$(M^\star, \mathsf{st}_\mathsf{A}) \leftarrow \mathsf{A}_0^{\mathsf{O}(K,\cdot)}(\kappa)$
$C_0^\star \leftarrow \mathsf{E}(K, M^\star)$
$C_1^\star \xleftarrow{\$} \mathcal{C}$
$b \xleftarrow{\$} \{0,1\}$
$b' \leftarrow \mathsf{A}_1^{\mathsf{O}(K,\cdot)}(\mathsf{st}_\mathsf{A}, C_b^\star)$
if $b' = b$ **then**
  return 1
else
  return 0

Figure 1: PCPA-security experiment. $\mathsf{O}(K,\cdot)$ is an encryption oracle which takes a plaintext $M \in \mathcal{M}$ as input and returns $\mathsf{E}(K,M) \in \mathcal{C}$.

2.2 Pseudorandom Functions (PRFs)

Let $\pi \coloneqq \{\pi_k : \{0,1\}^m \to \{0,1\}^{m'}\}_{k\in\{0,1\}^\kappa}$ be a family of functions, where m and m' are polynomial in κ.

Definition 1 (PRFs)**.** *π is said to be a PRF family if for any PPT algorithm D, there exists a negligible function $\mathsf{negl}(\kappa)$ such that:*

$$\left|\Pr\left[\mathsf{D}^{\pi_k(\cdot)}(\kappa) = 1 \mid k \xleftarrow{\$} \{0,1\}^\kappa\right] - \Pr\left[\mathsf{D}^{g(\cdot)}(\kappa) = 1 \mid g \xleftarrow{\$} \mathcal{G}\right]\right| < \mathsf{negl}(\kappa),$$

where $\mathcal{G}$ is a family of all functions that map an m-bit string to an m'-bit string. In particular, π is said to be a pseudorandom permutation (PRP) family if $m = m'$ and π is a family of bijections (then $\mathcal{G}$ turns to a permutation family).

2.3 Symmetric-Key Encryption (SKE)

As in [2, 3], we use SKE with a slightly-strong security notion.

Definition 2 (SKE)**.** *An SKE scheme Π_{SKE} consists of three-tuple non-interactive algorithms $\Pi_{\text{SKE}} \coloneqq (\mathsf{G}, \mathsf{E}, \mathsf{D})$, which are defined as follows:*

- *$K \leftarrow \mathsf{G}(\kappa)$: It is a probabilistic algorithm which takes a security parameter κ as input and outputs a secret key K.*
- *$C \leftarrow \mathsf{E}(K, M)$: It is an algorithm which takes a secret key $K \xleftarrow{\$} \{0,1\}^\kappa$ and a plaintext $M \in \mathcal{M}$ as input and outputs a ciphertext $C \in \mathcal{C}$, where $\mathcal{M}$ and $\mathcal{C}$ are sets of plaintexts and ciphertexts, respectively.*
- *$M \leftarrow \mathsf{D}(K, C)$: It is a deterministic algorithm which takes the secret key K and a ciphertext C as input and outputs a plaintext M or a special symbol $\perp$ which indicates decryption failure.*

We consider pseudorandomness against chosen plaintext attacks (PCPA security for short) [2, 3], which guarantees that ciphertexts are indistinguishable from random strings. We consider an experiment against an adversary $\mathsf{A} = (\mathsf{A}_0, \mathsf{A}_1)$ in Fig. 1.

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

$(\mathsf{DB}, \mathsf{st}_{\mathsf{D}}) \leftarrow \mathsf{D}_0(\kappa)$
$(k, \sigma^{(0)}, \mathsf{EDB}^{(0)}) \leftarrow \mathsf{Setup}(\kappa, \mathsf{DB})$
$\mathsf{st}_{\mathsf{D}} \leftarrow \mathsf{EDB}^{(0)}$
for $t = 1$ **to** Q **do**
  $q \leftarrow \mathsf{D}_t(\mathsf{st}_{\mathsf{D}})$
  $\langle(\sigma^{(t)}, \mathcal{X}_q^{(t-1)}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle \leftarrow \mathsf{Search}(k, q, \sigma^{(t-1)}; \mathsf{EDB}^{(t-1)})$
  $\mathsf{st}_{\mathsf{D}} \leftarrow (\mathsf{EDB}^{(t)}, \mathsf{trans}^{(t)})$
$b \leftarrow \mathsf{D}_{Q+1}(\mathsf{st}_{\mathsf{D}})$
return b

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

$(\mathsf{DB}, \mathsf{st}_{\mathsf{D}}) \leftarrow \mathsf{D}_0(\kappa)$
$(\mathsf{EDB}^{(0)}, \mathsf{st}_{\mathsf{S}}) \leftarrow \mathsf{S}_0(\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB}))$
$\mathsf{st}_{\mathsf{D}} \leftarrow \mathsf{EDB}^{(0)}$
for $t = 1$ **to** Q **do**
  $q \leftarrow \mathsf{D}_t(\mathsf{st}_{\mathsf{D}})$
  $\langle(\mathsf{st}_{\mathsf{S}}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle \leftarrow \mathsf{S}_t(\mathsf{st}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Srch}}(t, q); \mathsf{EDB}^{(t-1)})$
  $\mathsf{st}_{\mathsf{D}} \leftarrow (\mathsf{EDB}^{(t)}, \mathsf{trans}^{(t)})$
$b \leftarrow \mathsf{D}_{Q+1}(\mathsf{st}_{\mathsf{D}})$
return b

Figure 2: Real and ideal experiments.

Definition 3 (PCPA Security)**.** *Let Π_{SKE} be an SKE scheme. Π_{SKE} is said to be* PCPA*-secure if for any PPT algorithm* A*, we have:*

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{PCPA}}_{\Pi_{\text{SKE}},\mathsf{A}}(\kappa) = 1\right] - \frac{1}{2}\right| < \mathsf{negl}(\kappa).$$

PCPA-security can be achieved by common SKE schemes such as AES with counter mode. Note that one may employ CPA-secure SKE schemes for SSE-1, SSE-2, and our schemes, instead of PCPA-secure schemes, where CPA refers to (standard) chosen plaintext attacks.

2.4 Searchable Symmetric Encryption

Notations for SSE. Let λ and ℓ be polynomials in κ. Let $\Lambda := \{0,1\}^{\lambda}$ be a set of possible keywords (sometimes called a *dictionary*),[4] and $\mathcal{F}$ be a set of possible document files. We assume that each file $f_{\mathsf{id}} \in \mathcal{F}$ has the corresponding identifier $\mathsf{id} \in \{0,1\}^{\ell}$, which is irrelevant to the content of f_{id} (e.g., document numbers). As in previous works, we suppose that each file $f_{\mathsf{id}} := (\mathsf{id}, \mathcal{W}_{\mathsf{id}})$ consists of its identifier id and $\mathcal{W}_{\mathsf{id}} \subset \Lambda$, which is a set of distinct keywords contained in f_{id}. We sometimes write $f_i := (\mathsf{id}_i, \mathcal{W}_i)$ instead of $f_{\mathsf{id}_i} := (\mathsf{id}_i, \mathcal{W}_{\mathsf{id}_i})$ for simplicity. We consider a *global counter* t, which is initially set to zero, to describe a time-line for the SSE scheme. Namely, t is incremented for each search operation. A database DB is represented as a set of $(\mathsf{id}, \mathcal{W}_{\mathsf{id}})$, i.e., $\mathsf{DB} := \{(\mathsf{id}_i, \mathcal{W}_i)\}_{i=1}^{n}$, where n is the number of document files stored in the server. We denote by $\mathcal{W} := \bigcup_{i=1}^{n} \mathcal{W}_i$ a set of keywords in DB, and let $d := |\mathcal{W}|$. The size N of the database DB is defined by the number of (document, keyword) pairs in DB, i.e., $N := \sum_{i=1}^{n} |\mathcal{W}_i|$. Let ID be a set of identifiers in DB, i.e., $\mathsf{ID} := \{\mathsf{id} \mid (\mathsf{id}, \mathcal{W}_{\mathsf{id}}) \in \mathsf{DB}\}$. For any $w \in \Lambda$, ID_w represents a set of identifiers containing w in DB (i.e., $\mathsf{ID}_w := \{\mathsf{id} \mid \mathsf{id} \in \mathsf{ID} \wedge w \in \mathcal{W}_{\mathsf{id}}\}$).

Model. We define a syntax that captures both non-dynamic and dynamic SSE schemes since we deal with both in this paper. Note that this definition is essentially the same as those in [20, 22, 23] in the non-dynamic setting, and includes Curtmola et al.'s definition [2, 3] as a special case (see Appendix A for details). Unlike Curtmola et al.'s work [2, 3], we omit encryption and decryption algorithms for document files since it can be easily realized with any PCPA-secure (or CPA-secure) SKE scheme.

[4] One may consider an *unbounded* dictionary, i.e., $\Lambda = \{0,1\}^*$, by employing collision-resistant hash functions that map from arbitrary strings to λ-bit strings.

First of all, the client runs Setup with the security parameter κ and a database (i.e., a document-file set) DB, and gets a secret key k, state information $\sigma^{(0)}$, and an encrypted database $\mathsf{EDB}^{(0)}$. What the server stores is only $\mathsf{EDB}^{(0)}$. The client and the server run $\mathsf{Search} = (\mathsf{Search}_{\mathrm{C}}, \mathsf{Search}_{\mathrm{S}})$ to search a keyword $q \in \Lambda$ at t. The client (resp., the server) executes $\mathsf{Search}_{\mathrm{C}}$ with k, $\sigma^{(t)}$, and the keyword q (resp., $\mathsf{Search}_{\mathrm{S}}$ with $\mathsf{EDB}^{(t)}$), and obtains updated state information $\sigma^{(t+1)}$ and a search result $\mathcal{X}_q^{(t)}$ (resp., an updated encrypted database $\mathsf{EDB}^{(t+1)}$).

Definition 4 (SSE)**.** *An SSE scheme Σ over Λ consists of two-tuple algorithms $\Sigma := (\mathsf{Setup}, \mathsf{Search})$, which are defined as follows:*

- $(k, \sigma^{(0)}, \mathsf{EDB}^{(0)}) \leftarrow \mathsf{Setup}(\kappa, \mathsf{DB})$*: It is a non-interactive probabilistic algorithm which takes a security parameter κ and an initial database DB as input and outputs a secret key k, initial state information $\sigma^{(0)}$, and initial encrypted database $\mathsf{EDB}^{(0)}$.*
- $(\sigma^{(t+1)}, \mathcal{X}_q^{(t)}; \mathsf{EDB}^{(t+1)}) \leftarrow \mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$*: It is an interactive algorithm which consists of $\mathsf{Search}_{\mathrm{C}}$ and $\mathsf{Search}_{\mathrm{S}}$. $\mathsf{Search}_{\mathrm{C}}$ takes k, a keyword q to be searched, and $\sigma^{(t)}$ as input, and outputs updated state information $\sigma^{(t+1)}$ and a search result $\mathcal{X}_q^{(t)}$. $\mathsf{Search}_{\mathrm{S}}$ takes $\mathsf{EDB}^{(t)}$ as inputs and outputs $\mathsf{EDB}^{(t+1)}$.*

The correctness of the above model is defined as follows. Suppose that $\mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$ is executed for any $q \in \Lambda$ after t search operations for any t $(= \mathsf{poly}(\kappa))$. Then, Σ satisfies the correctness if the output $\mathcal{X}_q^{(t)}$ satisfies the following with overwhelming probability:

$$\mathcal{X}_q^{(t)} = \begin{cases} \mathsf{ID}_q & \text{if } q \in \mathcal{W}, \\ \emptyset & \text{if } q \notin \mathcal{W}. \end{cases}$$

Security. Following most previous works, we provide the simulation-based security definition for SSE. It is known that there is a trade-off between efficiency and security levels in SSE, and therefore we have to allow some leakage to perform efficient operations. Such information leakage is characterized as a *leakage function* $\mathcal{L} := (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Srch}})$. To put it briefly, $\mathcal{L}_{\mathsf{Setup}}$ and $\mathcal{L}_{\mathsf{Srch}}$ are information leaked during the setup and search operations, respectively.

We define adaptive security, which is a standard security notion for SSE. The notion is parameterized by a leakage function $\mathcal{L}$, and therefore it is called $\mathcal{L}$-adaptive security. Intuitively, $\mathcal{L}$-adaptive security guarantees that no information is leaked other than $\mathcal{L}$ even if an adversary adaptively performs update and search operations. Formally, we consider the following two experiments: a real experiment $\mathsf{Real}_{\mathsf{D}}$ between a PPT algorithm $\mathsf{D} = (\mathsf{D}_0, \mathsf{D}_1, \ldots, \mathsf{D}_{Q+1})$[5] and the client; and an ideal experiment $\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}$ between D and a simulator $\mathsf{S} = (\mathsf{S}_0, \ldots, \mathsf{S}_Q)$. The formal description of $\mathsf{Real}_{\mathsf{D}}$ and $\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}$ is given in Fig. 2.

Definition 5 ($\mathcal{L}$-adaptive security)**.** *Let Σ be an SSE scheme. Σ is said to be $\mathcal{L}$-adaptively secure if for any PPT algorithm D, there exists a PPT algorithm S such that:*

$$\left|\Pr\left[\mathsf{Real}_{\mathsf{D}}(\kappa, Q) = 1\right] - \Pr\left[\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}(\kappa, Q) = 1\right]\right| < \mathsf{negl}(\kappa).$$

Remark 1 (Non-Adaptive Security)**.** *If we consider real and ideal experiments where D_0 also outputs Q keywords at once and receives $\mathsf{st}_{\mathsf{D}} := (\mathsf{EDB}^{(t)}, \mathsf{trans}^{(t)})$ (and removes $\mathsf{D}_1, \ldots, \mathsf{D}_Q$), Def. 5 then turns to the definition of $\mathcal{L}$-non-adaptive security for SSE.*

[5] Q indicates the number of queries issued by D, and the upper bound of Q can be easily estimated from the running time of D.

	f_1	f_2	f_3	$\dots$	f_n
w_1	✓		✓	$\dots$	
w_2	✓	✓		$\dots$	✓
w_3		✓		$\dots$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$w_{\lvert\Lambda\rvert}$	✓			$\dots$	✓

(a) A look-up table.

Address	Value
$w_1 \| 1$	id_1
$w_1 \| 3$	id_3
$w_2 \| 1$	id_1
$\vdots$	$\vdots$
$w_{\lvert\Lambda\rvert} \| n$	id_n

(b) An index based on the look-up table.

Address	Value
$\pi_k(w_1 \| 1)$	id_1
$\pi_k(w_1 \| 3)$	id_3
$\pi_k(w_2 \| 1)$	id_1
$\vdots$	$\vdots$
$\pi_k(w_{\lvert\Lambda\rvert} \| n)$	id_n

(c) A secure index.

Figure 3: Curtmola et al.'s construction approach.

Concrete leakage functions for SSE-1 and SSE-2. Curtmola et al. [2, 3] considered the following specific leakage functions:[6]

- $\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB}) := (\Lambda, \{(\mathsf{id}_i, |f_i|)\}_{i=1}^n)$, where $\mathsf{DB} := \{(\mathsf{id}_i, \mathcal{W}_i)\}_{i=1}^n = \{f_i\}_{i=1}^n$.
- $\mathcal{L}_{\mathsf{Srch}}(t, q) := (t, \mathsf{SP}_q^{(t)}, \mathsf{AP}_q^{(t)})$, where $\mathsf{SP}_q^{(t)}$ and $\mathsf{AP}_q^{(t)}$ are defined as follows.
  - $\mathsf{SP}_q^{(t)}$ is a *search pattern* at t for q, which is a set of the global counters when the same keyword was previously searched, i.e., $\mathsf{SP}_q^{(t)} := \{t' \mid t' \in [t], \mathcal{L}_{\mathsf{Srch}}(t', q)\}$.
  - $\mathsf{AP}_q^{(t)}$ is an *access pattern* at t for q. It holds $\mathsf{AP}_q^{(t)} = \mathsf{ID}_q$ with overwhelming probability (depending on the correctness).

3 Forward-Index-Based SSE Scheme with the Efficient Dummy-Adding Procedure

As described in the introduction, the original SSE-2 contains bugs, which were already mentioned in [8]. In this section, we show that the existing fixes [8, 17] do not reflect Curtmola et al.'s construction idea. They require more dummies than the constructions really need and hence do not work well for a large dictionary (e.g., $|\Lambda| = \mathsf{exp}(\kappa)$). We then show how to fix them in the best way possible.

3.1 The SSE-2 Construction and Its Variants

First, we review the original SSE-2 scheme.

Construction idea. As described earlier, Curtmola et al. took the forward-index-based approach for SSE-2. At a high level, a secure index stored in the server should contain relations between each keyword $w \in \mathcal{W}$ and its corresponding files ID_w to perform the search operation correctly, however no information more than $\mathcal{L}_{\mathsf{Srch}}$ has to be leaked from the secure index by search.

Their basic construction idea consists of the following three steps (see Fig. 3 as an example): (1) create a look-up table describing relations between files and keywords in Fig. 3(a); (2) make an index in Fig. 3(b) based on the look-up table; and (3) use PRFs (or PRPs) to hide the relations from the server such as Fig. 3(c). The server cannot learn the relation between keywords and files from the secure index due to the underlying PRF π. On the one hand, the client can search arbitrary keyword $q \in \Lambda$ by computing a *trapdoor* $\mathcal{T}_q := (\pi_k(q \| 1), \dots, \pi_k(q \| n))$. The server can

[6] To be precise, Curtmola et al. did not consider the leakage in the form of leakage functions. Nevertheless, our definition captures the same leakage as in their papers [2, 3].

	Real part					Dummy part				
	f_1	f_2	f_3	$\dots$	f_n	f_1	f_2	f_3	$\dots$	f_n
w_1	✓		✓	$\dots$			✓		$\cdots$	✓
w_2	✓	✓		$\dots$	✓			✓	$\cdots$	
w_3		✓		$\dots$		✓		✓	$\cdots$	✓
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$w_{\lvert\Lambda\rvert}$	✓			$\dots$	✓		✓	✓	$\cdots$	

Figure 4: The horizontally-extended look-up table.

correctly return the search result by collecting values stored at each address in $\mathcal{T}_q$. However, the above idea is insufficient since the current form of the secure index leaks $|\mathcal{W}_1|, \dots, |\mathcal{W}_n|$, which can be extracted by counting the number of identifiers of each files in the secure index.

To prevent the server from learning the number of distinct keywords in each file, Curtmola et al. took the following approach: *adding dummy entries (i.e., pairs of a dummy address and file's identifier) to the secure index*. That is, for each file f_i, they tried to hide $|\mathcal{W}_i|$ by adding sufficiently many dummies. Specifically, Curtmola et al. introduced the concept of max, which is the maximum number of distinct keywords that can fit in the largest document file (i.e., $\mathsf{max} := \max_{i\in[n]}\{|f_1|, \dots, |f_n|\}$). Namely, for the largest file $f_{i^\star}$, $\mathsf{max} := c$ such that $\sum_{j=1}^{c} |w_j| \le |f_{i^\star}| < \sum_{j=1}^{c+1} |w_j|$, where $w_1 \le w_2 \le \cdots$ for $\Lambda = \{w_1, w_2, \dots\}$. Note that we set $\mathsf{max} := |\Lambda|$ if $\mathsf{max} > |\Lambda|$ since there are at most $|\Lambda|$ words.[7] Curtmola et al. showed that max is sufficient to hide $|\mathcal{W}_i|$ for each file f_i, , i.e., to simulate the setup procedure with only $\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB}) = (\Lambda, \{(\mathsf{id}_i, |f_i|)\}_{i=1}^{n})$. Specifically, for each file f_i, they added $(\mathsf{max} - |\mathcal{W}_i|)$ dummy entries to the secure index.

Bugs in the original SSE-2 and existing solutions. Although adding dummies was a good idea to guarantee the security, their dummy addition procedure had fateful flaws, which was already pointed out in [8], and hence the original SSE-2 construction does not work.[8] Therefore, we here describe a modified one based on [8, 17]. Roughly speaking, for each $w \in \Lambda$, each file f_i is associated with a real or dummy entry. Namely, if f_i contains w, a real entry for id_i is added to the secure index; otherwise, a dummy entry for id_i is added. Specifically, for each keyword $w \in \Lambda$ and each file f_i, an address $\mathtt{addr}$ is set as

$$\mathtt{addr} := \begin{cases} \pi_k(0\|w\|i) & \text{if } w \in \mathcal{W}_i, \\ \pi_k(1\|w\|i) & \text{if } w \notin \mathcal{W}_i. \end{cases}$$

Namely, the first bit indicates a real/dummy flag, and the secure index contains $|\Lambda|$ values for each file f_i. The above procedure can be viewed as a horizontally-extended version of the look-up table (Fig. 3(a)) in Fig. 4.

Let π be a PRP family, where $\pi := \{\pi_k : \{0,1\}^{\lambda+\lfloor\log n\rfloor+2} \to \{0,1\}^{\lambda+\lfloor\log n\rfloor+2}\}_{k\in\{0,1\}^\kappa}$, and Index be an array. The modified SSE-2 construction Σ'_{SSE2} based on the above "horizontally-extended" approach is given in Fig. 5.

Proposition 1 ([2, 8, 17])**.** *If π is a PRP family, an SSE scheme $\Sigma'_{\mathsf{SSE2}} = (\mathsf{Setup}, \mathsf{Search})$ constructed above is $\mathcal{L}$-adaptively secure, where*

$$\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB}) = (\Lambda, \{(\mathsf{id}_i, |f_i|)\}_{i=1}^{n}) \quad \textit{and} \quad \mathcal{L}_{\mathsf{Srch}}(t, q) = (\mathsf{SP}_q^{(t)}, \mathsf{AP}_q^{(t)}),$$

for any DB, any t, and any $q \in \Lambda$.

[7] Let us give an example in the case where each keyword consists of one-byte characters: for the largest file $f_{i^\star}$ with 100 KB, we can fit at most $\mathsf{max} = 51{,}328$ distinct keywords since we have $1 \cdot 2^8 + 2 \cdot 51{,}072 = 102{,}400$ bytes.

[8] For details, see Appendix B.

Σ'_{SSE2}: $\mathsf{Setup}(\kappa, \mathsf{DB})$

$k \xleftarrow{\$} \{0,1\}^\kappa$
for $i = 1$ **to** n **do**
  for $\forall w \in \Lambda$ **do**
    if $w \in \mathcal{W}_i$ **then**
      $\mathsf{Index}[\pi_k(0\|w\|i)] := \mathsf{id}$
    else
      $\mathsf{Index}[\pi_k(1\|w\|i)] := \mathsf{id}$ // add dummies
$\mathsf{EDB}^{(0)} := \mathsf{Index}$
return $(k, \sigma^{(0)} := \{n\}, \mathsf{EDB}^{(0)})$

Σ'_{SSE2}: $\mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$

Client:
for $i = 1$ **to** n **do**
  $\mathcal{T}_q^{(t)} \leftarrow \pi_k(0\|q\|i)$ // $\mathcal{T}_q^{(t)}$: trapdoor
Send $\mathsf{trans}_1^{(t)} := \mathcal{T}_q^{(t)}$ to the server
Server:
for $\forall \mathtt{addr} \in \mathcal{T}_q^{(t)}$ **do**
  if $\mathsf{Index}[\mathtt{addr}] \neq \mathrm{NULL}$ **then**
    $\mathcal{X}_q^{(t)} \leftarrow \mathsf{Index}[\mathtt{addr}]$ // $\mathcal{X}_q^{(t)}$: search result
Send $\mathsf{trans}_2^{(t)} := \mathcal{X}_q^{(t)}$ to the client
return $\mathsf{EDB}^{(t+1)} := \mathsf{Index}$
Client:
return $(\sigma^{(t+1)} := \sigma^{(t)}, \mathcal{X}_q^{(t)})$

Figure 5: The modified SSE-2 construction via the horizontally-extended approach.

3.2 Revisiting a Way of Adding Dummies

In the above scheme, a real or dummy entry is assigned to each keyword and each file. It means that the resulting secure index contains $n \cdot |\Lambda|$ entries, which is actually extremely large ($|\Lambda| = 2^\lambda$ in our setting). Therefore, we here consider a more efficient way of adding dummies.

Actually, Curtmola et al. could not reflect the concept of max in their construction correctly. Curtmola et al.'s idea is to add dummies to the secure index up to max, not $|\Lambda|$, for each file f_i. max seems sufficient to prevent the server from narrowing down candidates of the underlying file f_i. Therefore, our approach is to correctly employ and improve their idea for the construction.

The concept of max, reintroduced. First of all, how many dummy entries are necessary and sufficient to simulate the setup procedure with only $\{(\mathsf{id}_i, |f_i|)\}_{i=1}^n$ (i.e., $\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB})$)? As explained in the previous section, Curtmola et al. [2] partially answered it by introducing the concept of max, though they could not implemented the concept correctly.

We revisit the concept of max to reduce the number of dummy entries; we consider max for each file, not for the largest file, since max seems too much to hide $|\mathcal{W}_i|$ for all $i \in [n] \setminus \{i^\star\}$. max for a file f_i, which is denoted by $\mathsf{max}_{\mathsf{id}_i}$ (or max_i for simplicity), is the maximum number of keywords fit in f_i. Namely, for f_i, $\mathsf{max}_i := c$ such that $\sum_{j=1}^{c} |w_j| \leq |f_i| < \sum_{j=1}^{c+1} |w_j|$, where $w_1 \leq w_2 \leq \cdots$ for $\Lambda = \{w_1, w_2, \ldots\}$. Note that we set $\mathsf{max}_i := |\Lambda|$ if $\mathsf{max}_i > |\Lambda|$ since there are at most $|\Lambda|$ words.

Our idea is to add dummies to the secure index up to max_i, neither $|\Lambda|$ nor max, for each file f_i. By definition, it is clear that max_i is still sufficient to hide $|\mathcal{W}_i|$, i.e., to prevent the server from narrowing down candidates of the underlying file f_i. We show how to correctly employ our idea below.

Extending the look-up table vertically. We extend the look-up table in Fig. 3(a) vertically to add dummies up to max_i for each f_i (see Fig. 6). Then, the vertically-extended look-up table contains $\sum_{i=1}^n \mathsf{max}_i$ entries in total, whereas the horizontally-extended one contains $n \cdot |\Lambda|$. Note that max_i is (in general, significantly) smaller than $|\Lambda|$ for any $i \in [n]$, since $|f_i|$ (and hence max_i) is polynomial in κ while $|\Lambda|$ is exponential in κ in our setting. In any case, we have $\mathsf{max}_i \leq |\Lambda|$ for any file $f_i \in \mathcal{F}$. Specifically, we realize our approach, which is remarkably more efficient than the previous one [8, 17], by setting addresses for f_i in the form of $\pi_k(b\|w\|i)$, where $b \in \{0,1\}$ is a real/dummy flag, i.e., $b := 0$ for every $w \in \mathcal{W}_i$ and $b := 1$ for every $w \in [\mathsf{max}_i - |\mathcal{W}_i|]$. Indeed, our

		f_1	$\cdots$	f_i	$\cdots$	f_n
Real Part	w_1	✓	$\cdots$	✓	$\cdots$	
	w_2	✓	$\cdots$		$\cdots$	✓
	w_3		$\cdots$	✓	$\cdots$	
	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\ddots$	$\vdots$
	$w_{\lvert\Lambda\rvert}$	✓	$\cdots$		$\cdots$	✓
Dummy Part	1	✓	$\cdots$	✓	$\cdots$	✓
	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$
	$\mathsf{max}_n - \lvert\mathcal{W}_n\rvert$	✓	$\cdots$	✓	$\cdots$	✓
	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	
	$\mathsf{max}_1 - \lvert\mathcal{W}_1\rvert$	✓	$\cdots$	✓	$\cdots$	
	$\vdots$		$\cdots$	$\vdots$	$\ddots$	
	$\mathsf{max}_i - \lvert\mathcal{W}_i\rvert$		$\cdots$	✓	$\cdots$	

Figure 6: The vertically-extended look-up table.

Σ_{Ours}: $\mathsf{Setup}(\kappa, \mathsf{DB} = \{(\mathsf{id}_i, \mathcal{W}_i)\}_{i=1}^n)$

```
k ←$ {0,1}^κ
for i = 1 to n do
    for ∀w ∈ W_i do
        Index[π_k(0‖w‖i)] := id_i
    for β = 1 to max_i − |W_i| do
        Index[π_k(1‖β‖i)] := id_i
                    // Add dummies up to max_i
EDB^(0) := Index
return (k, σ^(0) := {n}, EDB^(0))
```

Figure 7: Our construction via the vertically-extended approach.

solution seems the most efficient fix for the SSE-2 construction.

3.3 Our Construction

Based on the above approach, we propose a new Setup algorithm in Fig. 7. Note that Search is the same as one in Section 3.1.

Theorem 1. *If π is a PRP family, a non-dynamic SSE scheme $\Sigma_{\mathsf{Ours}} = (\mathsf{Setup}, \mathsf{Search})$ constructed above is $\mathcal{L}$-adaptively secure, where*

$$\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB}) = (\Lambda, \{(\mathsf{id}_i, |f_i|)\}_{i=1}^n) \quad \text{and} \quad \mathcal{L}_{\mathsf{Srch}}(t, q) = (\mathsf{SP}_q^{(t)}, \mathsf{AP}_q^{(t)}),$$

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

Proof. We show how to construct the simulator S in $\mathsf{Ideal}_{\mathsf{D,S},\mathcal{L}}(\kappa, Q)$ as follows.

First, we show that S can simulate $\mathsf{EDB}^{(0)}$ by using $\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB})$, where $\mathsf{DB} = \{f_i\}_{i=1}^n = \{(\mathsf{id}_i, \mathcal{W}_i)\}_{i=1}^n$. In $\mathsf{Real}_{\mathsf{D}}(\kappa, Q)$, the client creates $\mathsf{EDB}^{(0)}$. For each file $f_i = (\mathsf{id}_i, \mathcal{W}_i)$, $\mathsf{EDB}^{(0)}$ contains max_i random strings associated with id_i. Therefore, roughly speaking, in $\mathsf{Ideal}_{\mathsf{D,S},\mathcal{L}}(\kappa, Q)$ the simulator S randomly chooses $(\sum_{i=1}^n \mathsf{max}_i)$ bit-strings, and for each file $i \in [n]$, assigns max_i strings as addresses for id_i . Then, D cannot distinguish the two experiments due to the security of π (see Def. 1).

Formally, for $\mathcal{L}_{\mathsf{Setup}}(\kappa, \{(\mathsf{id}_i, \mathcal{W}_i)\}_{i=1}^n) = (\Lambda, \{(\mathsf{id}_i, |f_i|)\}_{i=1})$, S simulates $\mathsf{EDB}^{(0)}$ as follows. Let $\mathcal{U}_i$ be a set of all addresses associated with id_i, and it is initialized as an empty set. For each $i \in [n]$, S computes max_i from $|f_i|$ and Λ, and randomly chooses max_i *unused* addresses. Namely, S repeats the following procedure max_i times:

1. $\mathtt{addr} \xleftarrow{\$} \{0,1\}^{\lambda + \lfloor \log n \rfloor + 2} \setminus \left(\bigcup_{j=1}^i \mathcal{U}_j\right)$.
2. $\mathcal{U}_i \leftarrow \mathtt{addr}$.
3. $\mathsf{Index}[\mathtt{addr}] := \mathsf{id}_i$.

Finally, S outputs $\mathsf{EDB}^{(0)} := \mathsf{Index}$. All addresses in $\mathsf{List}_{\mathsf{used}}$ are distinct from each other since π is a permutation, and look random due to the security of π. Hence, S can simulate $\mathsf{EDB}^{(0)}$ by only using $\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB})$.

We next show how to simulate the search procedure, i.e., how to simulate Search by only using $\mathcal{L}_{\mathsf{Srch}}(t, q)$. In $\mathsf{Real}_{\mathsf{D}}(\kappa, Q)$, the client first computes $\pi_k(0\|q\|i)$ for all $i \in [n]$, and sends the server $\mathsf{trans}_1^{(t)} = \mathcal{T}_q^{(t)} := \{\pi_k(0\|q\|1), \dots, \pi_k(0\|q\|n)\}$ as *trapdoors*. From the correctness, it holds $\mathsf{Index}[\pi_k(0\|q\|i)] = \mathsf{id}_i$ if f_i contains q; it holds $\mathsf{Index}[\pi_k(0\|q\|i)] = \text{NULL}$ otherwise. Moreover, it holds $\mathcal{X}_q^{(t)} = \mathsf{ID}_q$.

We construct S that simulates the above procedure correctly as follows. Let $\mathsf{List}_{\mathsf{trpdr}}$ be a list of all addresses that have been used for the response of search queries (i.e., used as trapdoors). We have to consider two cases depending on $\mathcal{L}_{\mathsf{Srch}}(t, q) = (\mathsf{SP}_q^{(t)}, \mathsf{AP}_q^{(t)})$:

(1) It is the first time to search for q, i.e., $\mathsf{SP}_q^{(t)} = \{t\}$.

(2) q has been queried before, i.e., $\mathsf{SP}_q^{(t)} \neq \{t\}$.

The reason why we consider the two cases is that trapdoors at the first search for a keyword q should be chosen at random, but those at subsequent searches should be the same as the first search.

(1) *It is the first time to search for q, i.e., $\mathsf{SP}_q^{(t)} = \{t\}$.* In $\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}(\kappa, Q)$, S simulates the above real procedures by inverse process. Note that S knows $\mathcal{L}_{\mathsf{Srch}}(t, q) = (\mathsf{SP}_q^{(t)}, \mathsf{AP}_q^{(t)})$. S randomly chooses an unused address $\mathtt{addr}_{i,q}$ as a trapdoor for q and id_i for all $i \in [n]$, but the domain from which $\mathtt{addr}_{i,q}$ is chosen depends on whether $\mathsf{id}_i \in \mathsf{AP}_q^{(t)}$ or not.

(1-a) Every identity $\mathsf{id}_i \in \mathsf{AP}_q^{(t)}$ should be stored at $\mathtt{addr}_{i,q}$. Note that S needs to avoid choosing addresses already used as trapdoors for other keywords contained in f_i. Therefore, S chooses $\mathtt{addr}_{i,q}$ from $\mathcal{U}_i \setminus \mathsf{List}_{\mathsf{trpdr}}$, where t' is a counter such that $(\mathsf{id}, t') \in \mathsf{List}_{\mathsf{id}}$.

(1-b) For every $\mathsf{id}_i \in \{\mathsf{id}_1, \dots, \mathsf{id}_n\} \setminus \mathsf{AP}_q^{(t)}$, the corresponding trapdoor should be an empty address. However, we have to pay attention to the fact that some empty addresses (i.e., addresses stored in $\mathsf{List}_{\mathsf{trpdr}}$) were assigned to trapdoors for other previously-searched keywords. Therefore, S randomly chooses $\mathtt{addr}_{i,q}$ from $\{0,1\}^{\lambda+\ell+1} \setminus (\mathsf{List}_{\mathsf{trpdr}} \cup \mathsf{List}_{\mathsf{addr}})$, where Let $\mathsf{List}_{\mathsf{addr}} := \bigcup_{i=1}^{n} \mathcal{U}_i$ be a list of all addresses used in $\mathsf{EDB}^{(0)}$.

S then adds $\mathtt{addr}_{i,q}$ to each of $\mathcal{T}_q^{(t)}$ and $\mathsf{List}_{\mathsf{trpdr}}$. Therefore, S can simulate the search procedure by setting $\mathsf{trans}_1^{(t)} := \mathcal{T}_q^{(t)}$ and $\mathsf{trans}_2^{(t)} := \mathsf{AP}_q^{(t)}$.

(2) *q has been queried before, i.e., $\mathsf{SP}_q^{(t)} \neq \{t\}$.* In this case, S has to create the same trapdoors as those at the first search. Therefore, S retrieves $\mathcal{T}_q^{(t')}$ for some $t' \in \mathsf{SP}_q^{(t)} \setminus \{t\}$, and sets $\mathsf{trans}_1^{(t)} := \mathcal{T}_q^{(t')}$ and $\mathsf{trans}_2^{(t)} := \mathsf{AP}_q^{(t)}$.

Thus, S can correctly simulate Search by only using $\mathcal{L}_{\mathsf{Srch}}(t, q)$. □

4 Extension to the Dynamic Setting

In this section, we show that our SSE scheme can be easily extended to the dynamic setting.

Additional notations for dynamic SSE. Since the underlying database and its corresponding keywords are changed by update operations, we will use the following notations in this section.

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

$(\mathsf{DB}^{(0)}, \mathsf{st}_{\mathsf{D}}) \leftarrow \mathsf{D}_0(\kappa)$
$(k, \sigma^{(0)}, \mathsf{EDB}^{(0)}) \leftarrow \mathsf{Setup}(\kappa, \mathsf{DB}^{(0)})$
$\mathsf{st}_{\mathsf{D}} \coloneqq \{\mathsf{EDB}^{(0)}\}$
for $t = 1$ **to** Q **do**
  $\mathsf{query} \leftarrow \mathsf{D}_t(\mathsf{st}_{\mathsf{D}})$
  if $\mathsf{query} = (\mathtt{upd}, \mathsf{op}, \mathsf{in})$ **then**
    $\langle(\sigma^{(t)}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle$
      $\leftarrow \mathsf{Update}(k, \mathsf{op}, \mathsf{in}, \sigma^{(t-1)}; \mathsf{EDB}^{(t-1)})$
  if $\mathsf{query} = (\mathtt{srch}, q)$ **then**
    $\langle(\sigma^{(t)}, \mathcal{X}_q^{(t-1)}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle$
      $\leftarrow \mathsf{Search}(k, q, \sigma^{(t-1)}; \mathsf{EDB}^{(t-1)})$
  $\mathsf{st}_{\mathsf{D}} \leftarrow (\mathsf{EDB}^{(t)}, \mathsf{trans}^{(t)})$
$b \leftarrow \mathsf{D}_{Q+1}(\mathsf{st}_{\mathsf{D}})$
return b

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

$(\mathsf{DB}^{(0)}, \mathsf{st}_{\mathsf{D}}) \leftarrow \mathsf{D}_0(\kappa)$
$(\mathsf{EDB}^{(0)}, \mathsf{st}_{\mathsf{S}}) \leftarrow \mathsf{S}_0(\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB}^{(0)}))$
$\mathsf{st}_{\mathsf{D}} \coloneqq \{\mathsf{EDB}^{(0)}\}$
for $t = 1$ **to** Q **do**
  $\mathsf{query} \leftarrow \mathsf{D}_t(\mathsf{st}_{\mathsf{D}})$
  if $\mathsf{query} = (\mathtt{upd}, \mathsf{op}, \mathsf{in})$ **then**
    $\langle(\mathsf{st}_{\mathsf{S}}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle$
      $\leftarrow \mathsf{S}_t(\mathsf{st}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Upd}}(t, \mathsf{op}, \mathsf{in}); \mathsf{EDB}^{(t-1)})$
  if $\mathsf{query} = (\mathtt{srch}, q)$ **then**
    $\langle(\mathsf{st}_{\mathsf{S}}; \mathsf{EDB}^{(t)}), \mathsf{trans}^{(t)}\rangle$
      $\leftarrow \mathsf{S}_t(\mathsf{st}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Srch}}(t, q); \mathsf{EDB}^{(t-1)})$
  $\mathsf{st}_{\mathsf{D}} \leftarrow (\mathsf{EDB}^{(t)}, \mathsf{trans}^{(t)})$
$b \leftarrow \mathsf{D}_{Q+1}(\mathsf{st}_{\mathsf{D}})$
return b

Figure 8: Real and ideal experiments for dynamic SSE.

In dynamic SSE, a global counter t is incremented for each update/search operation. A database $\mathsf{DB}^{(t)}$ at t is represented as a set of $(\mathsf{id}, \mathcal{W}_{\mathsf{id}})$, i.e., $\mathsf{DB}^{(t)} \coloneqq \{(\mathsf{id}_i, \mathcal{W}_i)\}_{i=1}^{n^{(t)}}$, where $n^{(t)}$ is the number of document files stored in the server at t. We denote by $\mathcal{W}^{(t)} \coloneqq \bigcup_{i=1}^{n^{(t)}} \mathcal{W}_i$ a set of keywords in $\mathsf{DB}^{(t)}$, and let $d^{(t)} \coloneqq |\mathcal{W}^{(t)}|$. The size of the database $\mathsf{DB}^{(t)}$ is defined by the number of (document, keyword) pairs in $\mathsf{DB}^{(t)}$, i.e., $N^{(t)} \coloneqq \sum_{i=1}^{n^{(t)}} |\mathcal{W}_i|$. Let $\mathsf{ID}^{(t)}$ be a set of identifiers in $\mathsf{DB}^{(t)}$, i.e., $\mathsf{ID}^{(t)} \coloneqq \{\mathsf{id} \mid (\mathsf{id}, \mathcal{W}_{\mathsf{id}}) \in \mathsf{DB}^{(t)}\}$. For any $w \in \Lambda$, $\mathsf{ID}_w^{(t)}$ represents a set of identifiers containing w in $\mathsf{DB}^{(t)}$ (i.e., $\mathsf{ID}_w^{(t)} \coloneqq \{\mathsf{id} \mid \mathsf{id} \in \mathsf{ID}^{(t)} \wedge w \in \mathcal{W}_{\mathsf{id}}\}$).

4.1 Model

We extend the syntax and security notions in Section 2.4 to the dynamic setting. Note that this extended definition is essentially the same as in [20, 22, 23].

Syntax. A dynamic SSE scheme Σ_{DSSE} over Λ consists of three-tuple algorithms $\Sigma_{\mathsf{DSSE}} \coloneqq (\mathsf{Setup}, \mathsf{Update}, \mathsf{Search})$, where Setup and Search are the same as those in Def. 4 and Update is defined as follows:

- $(\sigma^{(t+1)}; \mathsf{EDB}^{(t+1)}) \leftarrow \mathsf{Update}(k, \mathsf{op}, \mathsf{in}, \sigma^{(t)}; \mathsf{EDB}^{(t)})$: It is an interactive algorithm which consists of $\mathsf{Update}_{\mathrm{C}}$ and $\mathsf{Update}_{\mathrm{S}}$. $\mathsf{Update}_{\mathrm{C}}$, which is run by the client, takes k, an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, the corresponding input in (e.g., $\mathsf{in} = f_{\mathsf{id}}$ for add and $\mathsf{in} = \mathsf{id}$ for del), and $\sigma^{(t)}$ as input, and outputs updated state information $\sigma^{(t+1)}$. Similarly, $\mathsf{Update}_{\mathrm{S}}$, which is run by the server, takes $\mathsf{EDB}^{(t)}$ as inputs and outputs $\mathsf{EDB}^{(t+1)}$.

For simplicity, we consider the Update algorithm for a single document file. For example, the client runs Update m times when adding m files to the server. The correctness is defined in a similar way to the non-dynamic setting, so we omit to describe it here.

Adaptive security and forward privacy. We define $\mathcal{L}$-adaptive security for dynamic SSE by extending a leakage function $\mathcal{L}$ so that it includes a leakage function $\mathcal{L}_{\mathsf{Upd}}$ for updates.

To formalize $\mathcal{L}$-adaptive security, we consider similar experiments $\mathsf{Real}_{\mathsf{D}}$ and $\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}$ to non-dynamic ones in Fig. 2. The difference between the dynamic and non-dynamic versions is that

D can arbitrarily make update queries as well as search queries. The formal description of the experiments is given in Fig. 8.

Definition 6 ($\mathcal{L}$-Adaptive Security for Dynamic SSE)**.** *Let Σ_{DSSE} be a dynamic SSE scheme. Σ_{DSSE} is said to be $\mathcal{L}$-adaptively secure if for any PPT algorithm* D*, there exists a PPT algorithm* Sim *such that:*

$$|\Pr[\mathsf{Real}_{\mathsf{D}}(\kappa, Q) = 1] - \Pr[\mathsf{Ideal}_{\mathsf{D,S},\mathcal{L}}(\kappa, Q) = 1]| < \mathsf{negl}(\kappa).$$

Briefly speaking, *forward privacy* [23] guarantees that the adversary cannot learn if newly-added files contain previously-searched keywords. Therefore, we can say that forward privacy provides *genuinely secure* add operations. As explained in the introduction, Zhang et al. [24] showed that non-forward-private dynamic SSE schemes are vulnerable to the file injection attack, which is easy to carry out in the real world. Hence, forward privacy has become the minimum security requirement for dynamic SSE. Formally, forward privacy is defined as follows.

Definition 7 (Forward Privacy)**.** *Let Σ_{DSSE} be a $\mathcal{L}$-adaptively secure dynamic SSE scheme, where $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Upd}}, \mathcal{L}_{\mathsf{Srch}})$. Σ_{DSSE} is said to meet forward privacy if $\mathcal{L}_{\mathsf{Upd}}$ (for $\mathsf{op} = \mathsf{add}$) can be written as:*

$$\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{add}, \mathsf{in}) = \mathcal{L}'(t, \mathsf{add}, (\mathsf{id}, |\mathcal{W}_{\mathsf{id}}|, |f_{\mathsf{id}}|)),$$

where in *is input for a document file f_{id} and $\mathcal{L}'$ is a stateless function.*

Namely, Σ_{DSSE} is said to satisfy forward privacy if leaked information on addition only depends on identifiers and the number of keywords contained in the newly-added files (*not* keywords themselves).

Concrete leakage functions for our dynamic scheme. We consider the following specific leakage functions for updates.

- $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{add}, (\mathsf{id}, \mathcal{W}_{\mathsf{id}})) \coloneqq (\mathsf{id}, |f_{\mathsf{id}}|)$.
- $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{del}, \mathsf{id}) \coloneqq \mathsf{id}$.

Note that the above leakages are naturally derived by Curtmola et al.'s leakage function for setup, and clearly satisfy Def. 7, and thus, our scheme in the next section meets forward privacy.

4.2 Our Dynamic SSE Scheme

Based on our SSE scheme in Section 3.3, we propose a simple dynamic SSE scheme. The most appealing feature of our construction is that it does not require any state information. We believe that the *state-free* feature is quite important in the practical aspect since it allows the client to access the server via multiple devices without synchronization.[9] That is, an encrypted search service does not require the latest state information to search, and hence, the client only has to store an (initial) secret key k in their smartphone, laptop, and desktop computers for the service. Our dynamic SSE schemes are the first state-free constructions with forward privacy. To sum up, the forward-index-based approach conduces to the simple and state-free construction.

Let $\pi \coloneqq \{\pi_k : \{0,1\}^{\lambda+\ell+1} \to \{0,1\}^{\lambda+\ell+1}\}_{k \in \{0,1\}^\kappa}$ be a PRP family. We propose our dynamic SSE scheme $\Sigma_{\mathsf{DSSE}} = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Search})$ in Fig. 9 (we omit Setup since it is the same as our non-dynamic scheme).

[9]Though existing constructions can also achieve the state-free feature by encrypting and storing the state information on the server our construction is secure *even if* the state information is disclosed.

Σ_{DSSE}: $\mathsf{Update}(k, \mathsf{add}, (\mathsf{id}, \mathcal{W}_{\mathsf{id}}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$

Client:

1: **for** $\forall w \in \mathcal{W}_{\mathsf{id}}$ **do**
2: $\quad \mathcal{U}_{\mathsf{id}}^{(t)} \leftarrow \pi_k(0\|w\|\mathsf{id})$
3: **for** $\beta = 1$ **to** $\mathsf{max}_{\mathsf{id}} - |\mathcal{W}_{\mathsf{id}}|$ **do**
4: $\quad \mathcal{U}_{\mathsf{id}}^{(t)} \leftarrow \pi_k(1\|\beta\|\mathsf{id})$
5: Send $\mathsf{trans}_1^{(t)} := (\mathsf{id}, \mathcal{U}_{\mathsf{id}}^{(t)})$
6: **return** $\sigma^{(t+1)} := \emptyset$

Server:

7: $\mathcal{I} \leftarrow \mathsf{id}$
8: **for** $\forall \mathtt{addr} \in \mathcal{U}_{\mathsf{id}}^{(t)}$ **do**
9: $\quad \mathsf{Index}[\mathtt{addr}] := \mathsf{id}$
10: **return** $\mathsf{EDB}^{(t+1)} := (\mathcal{I}, \mathsf{Index})$

Σ_{DSSE}: $\mathsf{Update}(k, \mathsf{del}, \mathsf{id}, \sigma^{(t)}; \mathsf{EDB}^{(t)})$

Client:

1: Send $\mathsf{trans}_1^{(t)} := \mathsf{id}$
2: **return** $\sigma^{(t+1)} := \emptyset$

Server:

3: $\mathcal{I} := \mathcal{I} \setminus \{\mathsf{id}\}$
4: $\mathcal{A}_{\mathsf{id}} := \{\mathtt{addr} \mid \mathsf{Index}[\mathtt{addr}] = \mathsf{id}\}$
5: **for** $\forall \mathtt{addr} \in \mathcal{A}_{\mathsf{id}}$ **do**
6: $\quad \mathsf{Index}[\mathtt{addr}] := \mathrm{NULL}$
7: **return** $\mathsf{EDB}^{(t+1)} := (\mathcal{I}, \mathsf{Index})$

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

Client:

1: Send request (as $\mathsf{trans}_1^{(t)}$) to the server

Server:

2: Send $\mathsf{trans}_2^{(t)} := \mathcal{I}$ back to the client

Client:

3: **for** $\forall \mathsf{id} \in \mathcal{I}$ **do**
4: $\quad \mathcal{T}_q^{(t)} \leftarrow \pi_k(0\|q\|\mathsf{id})$
5: Send $\mathsf{trans}_3^{(t)} := \mathcal{T}_q^{(t)}$ to the server

Server:

6: **for** $\forall \mathtt{addr} \in \mathcal{T}_q^{(t)}$ **do**
7: $\quad$ **if** $\mathsf{Index}[\mathtt{addr}] \neq \mathrm{NULL}$ **then**
8: $\quad\quad \mathcal{X}_q^{(t)} \leftarrow \mathsf{Index}[\mathtt{addr}]$
9: Send $\mathsf{trans}_4^{(t)} := \mathcal{X}_q^{(t)}$ to the client
10: **return** $\mathsf{EDB}^{(t+1)} := (\mathcal{I}, \mathsf{Index})$

Client:

11: **return** $(\sigma^{(t+1)} := \emptyset, \mathcal{X}_q^{(t)})$

Figure 9: Our dynamic SSE construction.

Theorem 2. *If π is a PRP family, then a dynamic SSE scheme $\Sigma_{\mathsf{DSSE}} = (\mathsf{Setup}, \mathsf{Update}, \mathsf{Search})$ constructed above is $\mathcal{L}$-adaptively secure and forward-private, where*

$$\mathcal{L}_{\mathsf{Setup}}(\kappa, \mathsf{DB}) = (\Lambda, \{(\mathsf{id}_i, |f_i|)\}_{i=1}^n), \quad \mathcal{L}_{\mathsf{Upd}}(t, \mathsf{add}, (\mathsf{id}, \mathcal{W}_{\mathsf{id}})) = (\mathsf{id}, |f_{\mathsf{id}}|),$$
$$\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{del}, \mathsf{id}) = \mathsf{id}, \quad \mathcal{L}_{\mathsf{Srch}}(t, q) = (\mathsf{SP}_q^{(t)}, \mathsf{AP}_q^{(t)}),$$

for any $\mathsf{DB}^{(0)}$, any t and any $q \in \Lambda$.

We can prove this theorem in a way similar to Theorem 1. We give the proof in Appendix C.

Remark 2 (Degrading security levels for efficiency)**.** *To the best of our knowledge, all existing dynamic SSE schemes meet a weaker security than the above scheme. Especially, most schemes meet $\mathcal{L}$-adaptive security with $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{add}, (\mathsf{id}, \mathcal{W}_{\mathsf{id}})) = (\mathsf{id}, |\mathcal{W}_{\mathsf{id}}|, |f_{\mathsf{id}}|)$.*[10] *To improve efficiency, our scheme can be easily modified so that it is secure with such a leakage function. Namely, we can obtain a more efficient dynamic SSE scheme by just removing dummy-addition procedures (lines 3 and 4 of the addition procedure).*

[10]Such a leakage function still satisfies forward privacy.

Table 1: Enron email dataset

#files	#keywords	Size (KB)
517,401	214,874	$2,413,971$

Table 2: Statistical information on the dataset

parameter	max	min	average
$\lvert f_{\mathsf{id}} \rvert$ (bytes)	2,011,957	398	4,445
$\mathsf{max}_{\mathsf{id}}$	251,495	58	343.7
$\lvert \mathcal{W}_{\mathsf{id}} \rvert$	59,148	12	77.1
$\mathsf{max}_{\mathsf{id}} - \lvert \mathcal{W}_{\mathsf{id}} \rvert$	192,347	46	266.6

5 Implementation

We give a performance evaluation of the proposed schemes by C++ software implementation. We here implemented our dynamic construction and show that it seems sufficiently efficient in the practical sense. In particular, we show that the search procedure can be efficiently performed in practice, though it requires $\mathcal{O}(\mathsf{max}_{\mathsf{id}})$ computational cost in the asymptotic sense. Our experiments are done in Amazon EC2 using $\mathsf{m4.2xlarge}$ instance (32 GiB of memory and 8 CPU cores) with Ubuntu Server 18.04 LTS (HVM), EBS General Purpose (SSD) Volume Type. For the instantiation of a PRP π, we chose AES-GCM and GMAC to utilize Intel AES-NI instruction set. Throughout the experiment, we assume 128-bit security (key size). Our AES-GCM and GMAC implementation uses EVP functions API within OpenSSL library (version 1.1.0g). Although in our schemes (both of non-dynamic and dynamic ones), each operation (at the server side) can be parallelized, i.e., each server-side procedure of all algorithms can be executed in parallel, we implement our dynamic scheme on only a single thread.

Dataset. To create EDB, we deploy Enron Email dataset [25] (May 7, 2015 version) which is a well-known dataset containing roughly 2.4GB mail data from about 150 users, mostly senior management of Enron. Table 1 shows the number of files, the number of keywords, and the total size of Enron dataset. For the keywords used in EDB, we use only stems of the words that appear in the dataset. We obtain the set of keywords from dataset by applying the Porter stemming algorithm within NLTK (Natural Language ToolKit) library. We may add that to apply the stemming algorithm, we deleted the header information, symbols, and URL information in preprocessing. Table 2 shows the statistics on the size of each file, $\mathsf{max}_{\mathsf{id}}$ and $\lvert \mathcal{W}_{\mathsf{id}} \rvert$.

Addition/deletion cost. We show the setup/addition and deletion costs of our scheme in Figs. 10 and 11, respectively. Surprisingly, the figures show that the implementation of our scheme complete the whole of 517,401 data in roughly 75 seconds for addition and roughly two seconds for deletion, including communication time. Namely, the addition and deletion for our scheme take roughly 150μs and 3.8μs, respectively. It means the dummy-addition procedure does not significantly impair the performance, though our scheme should perform $\mathcal{O}(\mathsf{max}_{\mathsf{id}})$ computational cost.

Search cost. Fig. 12 shows the experimental result on search cost for our scheme. The vertical and horizontal axes show turn-around time for single search query, including communication cost, and the number of registered files, respectively. When 200,000 files are registered in EDB, our scheme requires roughly 0.7 seconds.

Thus, our forward-private scheme can be said that the performance is sufficiently practical depending on applications or datasets. Note that one should keep in mind that the communication cost is proportional to the total number of files. As noted at the beginning of this section, our implementation results are done on only single thread. Therefore, we can obtain much higher

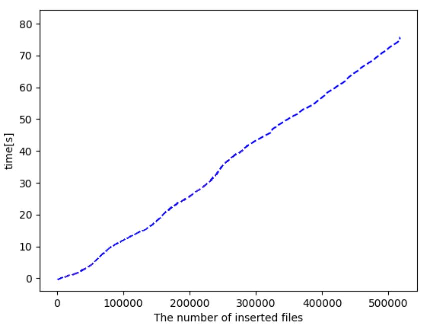

Figure 10: Setup/addition cost for our schemes.

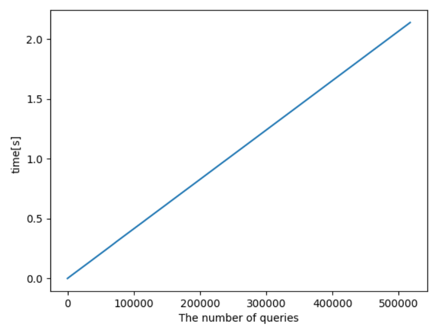

Figure 11: Deletion cost for our schemes.

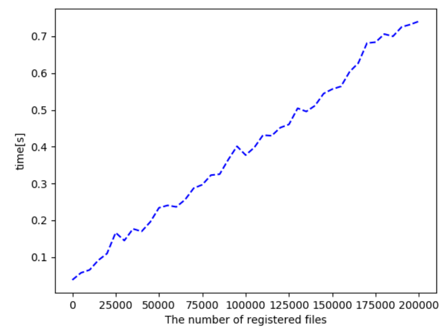

Figure 12: Search cost for our schemes.

performance by utilizing parallelization/vectorization.

6 How to Remove the Possibility of Search Error in SSE-1

Curtomola et al. proposed a non-adaptive secure SSE scheme called SSE-1 [2, 3]. This section shows SSE-1 has a problem that a search error occurs when the client searches a keyword not used in any stored files (but contained in the dictionary). We then propose a correction technique for this problem.

As in [2, 3], in this section, we assume that the dictionary size $|\Lambda|$ is polynomial in κ and that all keywords in Λ can be represented using at most v bits. Furthermore, we explain why SSE-1 requires the dictionary size is polynomial in κ and propose a way to improve SSE-1 to support the exponential-size dictionary.

6.1 The Original SSE-1 Construction and Its Problem

As described in the introduction, SSE-1 is constructed via the *inverted-index-based approach*. First of all, we review the construction idea of SSE-1.

The server holds an *array* Array and an *address table* Table as a secure index. The array consists of nodes $\{\mathsf{N}_{w,i}\}_{w\in\mathcal{W},i\in[|\mathsf{ID}_w|]}$. Each node $\mathsf{N}_{w,i}$ stores information about the i-th file identifier that includes w and is encrypted with a PCPA-secure SKE scheme. Decrypting all nodes for w, the server can obtain the correct search result ID_w. All nodes $\{\mathsf{N}_{w,1},\ldots,\mathsf{N}_{w,|\mathsf{ID}_w|}\}$ for each w are sequentially linked so that the server can decrypt all of them if the server is given the decryption key for $\mathsf{N}_{w,1}$. Each node $\mathsf{N}_{w,i}$ consists of the following three components:

- File identifier $\mathsf{id} \in \mathsf{ID}_w$
- Address of the next node $\mathsf{N}_{w,i+1}$
- Decryption key $k_{w,i+1}$ for the next node $\mathsf{N}_{w,i+1}$

The server can obtain an address and a decryption key for $\mathsf{N}_{w,i+1}$ in addition to $\mathsf{id} \in \mathsf{ID}_w$ by decrypting $\mathsf{N}_{w,i}$. Namely, $\{\mathsf{N}_{w,i}\}_{i\in[|\mathsf{ID}_w|]}$ is sequentially decrypted.

Table stores information about an address a_w and a decryption key $k_{w,1}$ for the first node $\mathsf{N}_{w,1}$ of each keyword $w \in \Lambda$ in each row (see Fig. 13). Note that each row is encrypted. When the client searches for w, the client sends the server a row number $\pi_{K_3}(w)$ and a decryption key $f_{K_2}(w)$

Row No.	Value
$\pi_{K_3}(w_1)$	$(a_{w_1}\|k_{w_1,1}) \oplus f_{K_2}(w_1)$
$\pi_{K_3}(w_2)$	$(a_{w_2}\|k_{w_2,1}) \oplus f_{K_2}(w_2)$
$\pi_{K_3}(w_3)$	$(a_{w_3}\|k_{w_3,1}) \oplus f_{K_2}(w_3)$
$\vdots$	$\vdots$
$\pi_{K_3}(w_{\|\Lambda\|})$	$(a_{w_{\|\Lambda\|}}\|k_{w_{\|\Lambda\|},1}) \oplus f_{K_2}(w_{\|\Lambda\|})$

Figure 13: An Address Table Table.

for the row. As a result, the server obtains an address a_w and a decryption key $k_{w,1}$ for $\mathsf{N}_{w,1}$ from Table. The server then decrypts $\mathsf{N}_{w,1}$ in Array and gets an address and a decryption key of the next node $\mathsf{N}_{w,2}$, in addition to $\mathsf{id} \in \mathsf{ID}_w$. Namely, the server can initiate the sequential decryption of $\{\mathsf{N}_{w,i}\}_{i\in[|\mathsf{ID}_w|]}$ by getting the address and decryption key of $\mathsf{N}_{w,1}$ from Table.

The server can finally obtain ID_w by repeating the sequential decryption. The address and decryption key in the last node are set to all zeros, which indicates "termination."

There are three points to keep in mind when Array and Table are created.

(a) If Array consists of only $\{\mathsf{N}_{w,i}\}_{w\in\mathcal{W},i\in[|\mathsf{ID}_w|]}$ then the database size $N = \sum_{i=1}^{n} |\mathcal{W}_i|$, leaks to the server from the number of nodes.

(b) If the server knows the relationship between keywords and rows in Table, then the server can identify the keyword corresponding to a search query.

(c) If Table consists of *only* keywords in $\mathcal{W}$, then $|\mathcal{W}|$ leaks to the server from the number of rows of Table.

We first explain (a). Note that N holds information about the number of distinct keywords in the stored files, which is not included in the leakage function described in Section 2.4. It must be kept secret from the server. However, N leaks to the server from Array since $|\{\mathsf{N}_{w,i}\}_{w\in\mathcal{W},i\in[|\mathsf{ID}_w|]}| = N$. Thus, sufficiently-many dummy nodes need to be added to Array to hide N from the server.

The number of dummy nodes to be created is calculated using max_i described in Section 3.2. Recall that max_i is the maximum number of distinct keywords that f_i can contain, and it can be calculated from the stored files by the server. Creating $\mathsf{max}_i - |\mathcal{W}_i|$ dummy nodes can hide $|\mathcal{W}_i|$ from the server, since the server cannot determine how many nodes are dummy in max_i nodes. The client can hide N from the server by applying this technique to all files and summing them up. The reason is the following: note that the number of nodes including dummy nodes in Array is $\mathsf{max}_{\mathsf{DB}} := \sum_{i=1}^{n} \mathsf{max}_i$, since the number of dummy nodes is $\sum_{i=1}^{n}(\mathsf{max}_i - |\mathcal{W}_i|) = \sum_{i=1}^{n} \mathsf{max}_i - \sum_{i=1}^{n} |\mathcal{W}_i| = \sum_{i=1}^{n} \mathsf{max}_i - N$. Then, $\mathsf{max}_{\mathsf{DB}}$ can be calculated by the server from the stored files. As a result, N can be hidden from the server. The rows in Table are randomly permuted for (b), and $|\Lambda| - |\mathcal{W}|$ dummy rows are created in Table for (c).

SSE-1 uses a PRF and two PRPs with the following parameters:

- $f : \{0,1\}^\kappa \times \{0,1\}^v \to \{0,1\}^{s+\kappa}$,
- $\pi : \{0,1\}^\kappa \times \{0,1\}^v \to \{0,1\}^v$,
- $\psi : \{0,1\}^\kappa \times \{0,1\}^s \to \{0,1\}^s$,

where $s := \lceil \log_2(\mathsf{max}_{\mathsf{DB}}) \rceil$ is the bit length of each node address. Each row in Table is masked with $f_{K_2}(w)$, where $K_2 \in \{0,1\}^\kappa$ and $w \in \mathcal{W}$. The randomization of the row order is executed by using $\pi_{K_3}(w)$, where $K_3 \in \{0,1\}^\kappa$ and $w \in \mathcal{W}$. ψ is used to determine node addresses in Array. Let $\Pi_{\text{SKE}} = (\mathsf{G}, \mathsf{E}, \mathsf{D})$ be a PCPA-secure SKE scheme and $\mathsf{ID}_w = \{\mathsf{id}_{w,1}, \mathsf{id}_{w,2}, \ldots, \mathsf{id}_{w,|\mathsf{ID}_w|}\}$ for all $w \in \mathcal{W}$.

Σ_{SSE1}: $\mathsf{Setup}(\kappa, \mathsf{DB})$

Building Array**:**

$K_1 \xleftarrow{\$} \{0,1\}^\kappa$
$\mathsf{ctr} = 1$
for $\forall w \in \mathcal{W}$ **do**
  $k_{w,1} \leftarrow \mathsf{G}(\kappa)$
  for $i = 1$ **to** $|\mathsf{ID}_w|$ **do**
    $k_{w,i+1} \xleftarrow{\$} \mathsf{G}(\kappa)$
    if $i < |\mathsf{ID}_w|$ **then**
      Create a node $\mathsf{N}_{w,i} := (\mathsf{id}_{w,i} \| \psi_{K_1}(\mathsf{ctr}+1) \| k_{w,i+1})$
    else
      Create a last node $\mathsf{N}_{w,|\mathsf{ID}_w|} := (\mathsf{id}_{w,|\mathsf{ID}_w|} \| 0^{s+\kappa})$
    $\mathsf{Array}[\psi_{K_1}(\mathsf{ctr})] := \mathsf{E}(k_{w,i}, \mathsf{N}_{w,i})$
    if $i = 1$ **then**
      $a_w := \psi_{K_1}(\mathsf{ctr})$ // Address of $\mathsf{N}_{w,1}$
    $\mathsf{ctr} = \mathsf{ctr} + 1$
for $j = 1$ **to** $\mathsf{max_{DB}} - N$ **do**
  $\mathsf{Array}[\psi_{K_1}(\mathsf{ctr})] \xleftarrow{\$} \{0,1\}^{\ell+s+\kappa}$ // Create a dummy node
  $\mathsf{ctr} = \mathsf{ctr} + 1$

Building Table**:**

$K_2, K_3 \xleftarrow{\$} \{0,1\}^\kappa$
for $\forall w \in \mathcal{W}$ **do**
  $\mathsf{Table}[\pi_{K_3}(w)] := (a_w \| k_{w,0}) \oplus f_{K_2}(w)$
  $\Pi_{\mathcal{W}} \leftarrow \pi_{K_3}(w)$ // $\Pi_{\mathcal{W}}$: set of row numbers for $w \in \mathcal{W}$
for $\forall w \in \Lambda \setminus \mathcal{W}$ **do**
  $v_w \xleftarrow{\$} \{0,1\}^v \setminus \Pi_{\mathcal{W}}$, and $c_w \xleftarrow{\$} \{0,1\}^{s+\kappa}$
  $\mathsf{Table}[v_w] := c_w$ // Dummy rows for $w \notin \mathcal{W}$
return $(k := (K_2, K_3), \sigma^{(0)} := \varepsilon, \mathsf{EDB}^{(0)} := (\mathsf{Array}, \mathsf{Table}))$

Σ_{SSE1}: $\mathsf{Search}((K_2, K_3), q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$

Client:

1: $\mathcal{T}_q^{(t)} \leftarrow (\pi_{K_3}(q), f_{K_2}(q))$ // $\mathcal{T}_q^{(t)}$: trapdoor
2: Send $\mathsf{trans}_1^{(t)} := \mathcal{T}_q$ to the server

Server:

3: Parse $\mathsf{trans}_1^{(t)}$ as (γ_1, γ_2)
4: **if** $\mathsf{Table}[\gamma_1] =$ NULL **then**
5:   Set $\mathcal{X}_q^{(t)} := \phi$ and go to line 12
6: Parse $\mathsf{Table}[\gamma_1] \oplus \gamma_2$ as (a'_1, k'_1)
7: Set $i = 1$
8: **while** $(a'_i \| k'_i) \neq 0^{s+\kappa}$ **do**
9:   Parse the result of $\mathsf{D}(k'_i, \mathsf{Array}[a'_i])$ as $(\mathsf{id}', a'_{i+1}, k'_{i+1})$
10:   $\mathcal{X}_q^{(t)} \leftarrow \mathsf{id}'$ // $\mathcal{X}_q^{(t)}$: search result
11:   $i = i + 1$
12: Send $\mathsf{trans}_2^{(t)} := \mathcal{X}_q^{(t)}$ to the client

Client:

13: **return** $(\sigma^{(t+1)} := \sigma^{(t)}, \mathcal{X}_q^{(t)})$

Figure 14: The original SSE-1 scheme.

We formally describe the original SSE-1 in Fig. 14 (See the operation example of SSE-1 in Appendix D).

The problem of the original SSE-1 construction. We show that SSE-1 causes a search error violating the correctness of the search procedure when the client searches for $w' \notin \mathcal{W}$. In fact, Curtmola et al. [2, 3] did not discuss the case where the client searches for such a keyword.[11] However, it is necessary to consider the case in practice since the client is likely to search for keywords that do not appear in any files.[12]

There are three kinds of search results as follows.

i) The server sends the correct search result to the client. (This case satisfies the correctness.)
ii) The server sends an incorrect search result to the client. (This case violates the correctness.)
iii) The server does not send the search result to the client for some reason. (This case viorates the correctness.)

Hereafter, we analyze the processes when the client searches for $w' \notin \mathcal{W}$. Denote the value of undefined rows in Table and undefined addresses in Array as NULL. We summarize the following discussions in Fig. 15.

[11] Strictly speaking, lines 4–5 of the search procedure captrure the case where the client searches for $w' \notin \mathcal{W}$. However, the process is not a sufficient countermeasure for such a case since $\mathsf{Table}[\pi_{K_3}(w')]$ for $w' \notin \mathcal{W}$ does not always be NULL, as shown in this section.

[12] Note that SSE-2 correctly performs the search procedure even when the client searches for $w' \notin \mathcal{W}$ since the corresponding (dummy) addresses of the secure index are set to the values that are never accessed in the search procedure.

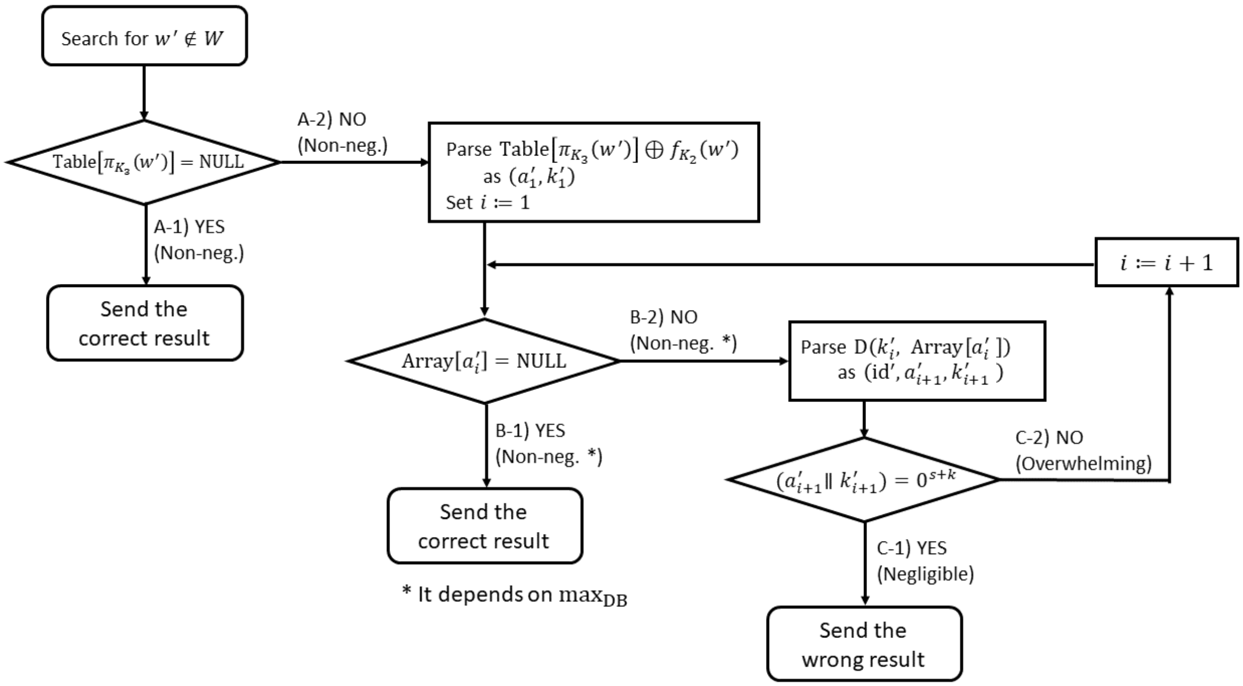

Figure 15: The process flow of searching for $w' \notin \mathcal{W}$. (Note that the infinite loop occurs with non-negligible probability.)

Let consider the case where the client searches for $w' \notin \mathcal{W}$. The client sends $(\pi_{K_3}(w'), f_{K_2}(w'))$, and the server checks the value of $\mathsf{Table}[\pi_{K_3}(w')]$. Then, there are two possible cases.

A-1) $\mathsf{Table}[\pi_{K_3}(w')] = \text{NULL}$, and then the server aborts the search process.

A-2) $\mathsf{Table}[\pi_{K_3}(w')] \neq \text{NULL}$, and then the server executes $\mathsf{Table}[\pi_{K_3}(w')] \oplus f_{K_2}(w')$.

In the case A-1), the server can identify that there is no file identifier to send and can send the empty set as the search result. Thus, this case satisfies the correctness.

In the case A-2), the result of $\mathsf{Table}[\pi_{K_3}(w')] \oplus f_{K_2}(w')$ becomes a random value since the dummy row is randomly chosen at line 16 in the setup procedure. However, the server parses the random value as (a'_1, k'_1), according to the procedure at line 4 in the search procedure, since it cannot identify whether the row is a dummy. The probability of the case A-2) is $(|\Lambda| - |\mathcal{W}|)/(2^v - |\mathcal{W}|)$ and non-negligible since v, which is the bit length of row number, is about $\log_2|\Lambda|$.

The case A-2) is divided into two cases depending on a'_1.

B-1) $\mathsf{Array}[a'_1] = \text{NULL}$, and then the server aborts the search process.

B-2) $\mathsf{Array}[a'_1] \neq \text{NULL}$, and then the server executes $\mathsf{D}(k'_1, \mathsf{Array}[a'_1])$.

In the case B-1), the server can send the empty set as the search result since it can identify that there is no file to send. Thus, this case satisfies the correctness.

In the case B-2), the decryption result becomes a random value since k'_1 is the incorrect key. The server parses the random value as $(\mathsf{id}', a'_2, k'_2)$, according to the procedure at line 7 in the search procedure.

The probability of B-1) depends on the number of nodes in Array. Recall that the bit length of the node address is $s := \lceil \log_2(\mathsf{max_{DB}}) \rceil$ where $\mathsf{max_{DB}}$ is the number of nodes. Thus, the probability of B-1) is maximized in the case where $\mathsf{max_{DB}} = 2^{s-1}+1$, and the probability of B-1) is $1-\mathsf{max_{DB}}/2^s$, which is about $1/2$. On the other hand, if $\mathsf{max_{DB}} = 2^s$, the probability of B-1) is $1 - \mathsf{max_{DB}}/2^s$, which is 0, that is, B-2) is always occurs.

After the case B-2) occurs, there are two cases depending on a'_2 and k'_2.

C-1) $(a'_2 \| k'_2) = 0^{s+\kappa}$, and then the server terminates the search process.

C-2) $(a'_2 \| k'_2) \neq 0^{s+\kappa}$, and then the server continues the search process.

In the case C-1), the server recognizes that the node is the last one since the search process follows the regular procedure. The server sends the client id' obtained by the incorrect decryption without noticing the error. Thus, this case violates the correctness that corresponds to the case of ii). Fortunately, the probability of the case of C-1) is negligibly small since it is $1/2^{s+\kappa}$.

After the case C-2) occurs, either B-1) or B-2) occurs again, depending on a'_2. Note that C-2) occurs with overwhelming probability, i.e., the search process almost never ends with C-1). Unfortunately, the loop of the above erroneous search process might occur since, as explained above, B-2) occurs with non-negligible probability. In particular, if $\mathsf{max_{DB}} = 2^s$, i.e., the probability of B-1) is zero, the server repeats the search process almost infinitely since the server cannot terminate the search process except for negligible probability. As a result, the server cannot send the search results to the client. This case violates the correctness since it corresponds to the case of iii).

6.2 How to Fix the Search Error

This section shows how to fix the search error in the previous section.

Solution 1. First, we propose a technique to stop the search loop by detecting the error without changing the setup procedure. When the search loop occurs, some unusual procedures are performed. For instance, the file identifier obtained by decrypting a node with an incorrect key may not actually exist in ID. Thus, the server can detect the error by checking whether the obtained file identifier is included in ID (if the server knows the identifiers corresponding to the stored files).

Also, the number of nodes decrypted in a search process is at most n, which is the total number of files. If more than n nodes are decrypted in the search process, the server can detect the search error.

Although the server can always abort the loop with the above detection ideas, the redundant search process still occurs. Thus, we next consider more efficient techniques to eliminate the error by modifying the setup procedure.

Solution 2. Intuitively, the search error can be resolved by changing the row numbers of dummies to be generated from $v_w \xleftarrow{\$} \{0,1\}^v \setminus \Lambda$. Namely, line 21 in the setup procedure is changed to the following process:

$$v_w \xleftarrow{\$} \{0,1\}^v \setminus \Pi_\Lambda, \text{and } c_w \xleftarrow{\$} \{0,1\}^{s+\kappa},$$

where $\Pi_\Lambda := \{x \in \{0,1\}^v : w \in \Lambda, x = \pi_{K_3}(w)\}$. Note that it is necessary to enlarge v when the size of $\{0,1\}^v \setminus \Pi_\Lambda$ is not enough for all $w' \notin \mathcal{W}$. Although this solution avoids the search error, the client needs to calculate $\pi_{K_3}(w')$ for all w' such that $w' \in \Lambda \wedge w' \notin \mathcal{W}$ in the setup procedure. This puts a big burden on the client. Thus, we propose another more efficient solution.

Solution 3. We show the search error can be resolved by just adding a special node to Array. For unused keywords, we prepare the special node N', which is a zero string, to indicate that there is no file identifier that should be sent to the client. More formally, the following procedure is inserted at the end of "Building Array":

$$k' \xleftarrow{\$} \mathsf{G}(\kappa)$$
Create a special node $\mathsf{N}' := 0^{\ell+s+\kappa}$
$$\mathsf{Array}[\psi_{K_1}(\mathsf{ctr})] := \mathsf{E}(k', \mathsf{N}')$$
$$\alpha := \psi_{K_1}(\mathsf{ctr})$$

Note that the client must create N' even if $\Lambda = \mathcal{W}$, i.e., even if there is no unused keyword in the dictionary, since the server can identify whether $\Lambda = \mathcal{W}$ or not from the number of nodes in Array. Thus, the number of nodes in Array always be $\mathsf{max}_{\mathsf{DB}} + 1$.

Also, in Table, we change all rows for all $w' \notin \mathcal{W}$ to point to the N', not just random value. As a result, when the client searches for $w' \notin \mathcal{W}$, the server always decrypts N' and understands there is no file to return to the client. More formally, lines 21–22 in the setup procedure are changed to the following process:

$$\mathsf{Table}[\pi_{K_3}(w)] := (\alpha \| k') \oplus f_{K_2}(w)$$

Finally, we show that $\mathsf{SSE\text{-}1}$ with Solution 3 is ($\mathcal{L}$-non-adaptively) secure. We show that our modifications only require minor changes to the original proof in [2, 3].

We first check the modified Array. Our modification does not change nodes other than the special node N', so that we discuss how to simulate N'. Noting that the procedure of creating N' does not depend on the stored file, the simulator also can run the procedure. Then, the simulator performs $k' \xleftarrow{\$} \{0,1\}^\kappa$ and $\alpha' \xleftarrow{\$} \{0,1\}^s$ instead of $k' \xleftarrow{\$} \mathsf{G}(\kappa)$ and $\psi_{K_1}(\mathsf{ctr})$, respectivley.

We next check the modified Table. Each row for $w' \notin \mathcal{W}$ is encrypted with $f(w')$ like the rows for $w \in \mathcal{W}$, and thus, these rows can be simulated in the same way as the rows for $w \in \mathcal{W}$. Namely, roughly speaking, the simulation can be done as follows: Let Q' be the number of distinct queries in $\Lambda/\mathcal{W}$, which the simulator can identify from the access pattern and the search pattern. Then, for $i \in [Q']$, the simulator sets $(\alpha', k') \oplus r_i$ as a value of the corresponding row, where $r_i \leftarrow \{0,1\}^{s+\kappa}$.

6.3 Toward Handling the Exponential-Size Dictionary

$\mathsf{SSE\text{-}1}$ assumes that the dictionary size $|\Lambda|$ is polynomial in κ since it creates the address table for all $w \in \Lambda$ in order to hide $|\mathcal{W}|$ from the server. However, if the total size of the stored files is smaller than $|\Lambda|$, it is not necessary to create rows for all $w \in \Lambda$. This is because the server knows that all keywords in Λ cannot fit in the stored files. Based on this fact, $\mathsf{SSE\text{-}1}$ can be improved to support the case where the dictionary size is exponential in κ by introducing the similar idea of max, described in Section 3.1.

Let $\mathsf{max}' := c$ such that $\sum_{j=1}^{c} |w_j| \leq \sum_{j=1}^{n} |f_j| < \sum_{j=1}^{c+1} |w_j|$, where $w_1 \leq w_2 \leq \cdots$ for $\Lambda = \{w_1, w_2, \ldots\}$. max' means the maximum number of distinct keywords that all stored files can contain.[13] Thus, we can make Table whose number of rows is max'. Note that max' does not leak any useful information to the server since the server also can compute it from the stored (encrypted) files. The rows for $w \in \mathcal{W}$ are created in the same way in the original $\mathsf{SSE\text{-}1}$. Also, the remaining $\mathsf{max}' - |\mathcal{W}|$ rows are created to connect the special node N' as in Solution 3.

[13] More precisely, the maximum number of distinct keywords contained in all stored files should be calculated from *each* file size, not the sum of them. In other words, max' may be greater than it. Nonetheless, we adopt the definition of max' for simplicity since it is sufficient to hide $|\mathcal{W}|$ and is independent of the size of Λ.

7 Conclusion

This paper focused on Curtmola et al.'s seminal work [2, 3] in the SSE area and revealed two previously-overlooked problems in their SSE constructions, SSE-1 and SSE-2. First, we pointed out that SSE-2 and its variants [8, 17] did not appropriately implement Curtmola et al.'s construction idea of dummy-addition procedures, and proposed a new construction based on SSE-2 that provides the smaller secure index than the above schemes. We further showed that our scheme can be easily extended to a dynamic version and is practically efficient via our software implementation. Second, we pointed out that SSE-1 violates the search correctness since the search for keywords that do not appear in any stored files triggers unexpected behavior. We demonstrated that the error can be detected by the server, the detection procedure, of course, should be implemented in advance. Besides, we showed how to fix the error, and how to extend SSE-1 to handle the exponential-sized dictionary.

Acknowledgments. We would like to thank Takato Hirano and Yutaka Kawai for their valuable comments. This work was supported by JSPS KAKENHI Grant Numbers JP21H03441, JP21H03395, JP20J21248, JP18H05289, JP18H03238, JP18K11293, and JP17H01752, and MEXT Leading Initiative for Excellent Young Researchers.

References

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

[2] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," ACM SIGSAC Conference on Computer and Communications Security, CCS 2006, New York, NY, USA, pp.79–88, ACM, 2006.

[3] R. Curtmola, J.A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," Journal of Computer Security, vol.19, no.5, pp.895–934, 2011.

[4] G. Asharov, G. Segev, and I. Shahaf, "Tight tradeoffs in searchable symmetric encryption," Advances in Cryptology – CRYPTO 2018, ed. H. Shacham and A. Boldyreva, Cham, pp.407–436, Springer International Publishing, 2018.

[5] G. Asharov, M. Naor, G. Segev, and I. Shahaf, "Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations," ACM Symposium on Theory of Computing, STOC 2016, New York, NY, USA, pp.1101–1114, ACM, 2016.

[6] D. Cash and S. Tessaro, "The locality of searchable symmetric encryption," Advances in Cryptology – EUROCRYPT 2014, ed. P. Nguyen and E. Oswald, Lecture Notes in Computer Science, vol.8441, pp.351–368, Springer Berlin Heidelberg, 2014.

[7] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," Advances in Cryptology – CRYPTO 2018, ed. H. Shacham and A. Boldyreva, Cham, pp.371–406, Springer International Publishing, 2018.

[8] K. Kurosawa and Y. Ohtaki, "UC-secure searchable symmetric encryption," Financial Cryptography and Data Security, FC 2012, ed. A.D. Keromytis, Berlin, Heidelberg, pp.285–298, Springer Berlin Heidelberg, 2012.

[9] K. Kurosawa and Y. Ohtaki, "How to update documents verifiably in searchable symmetric encryption," Cryptology and Network Security, CANS 2013, ed. M. Abdalla, C. Nita-Rotaru, and R. Dahab, Cham, pp.309–328, Springer International Publishing, 2013.

[10] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," ACM SIGSAC Conference on Computer and Communications Security, CCS 2012, New York, NY, USA, pp.965–976, ACM, 2012.

[11] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," Financial Cryptography and Data Security, FC 2013, ed. A.R. Sadeghi, Berlin, Heidelberg, pp.258–274, Springer Berlin Heidelberg, 2013.

[12] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," ACM SIGSAC Conference on Computer and Communications Security, CCS 2014, New York, NY, USA, pp.310–320, ACM, 2014.

[13] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.C. Roşu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," Network and Distributed System Security Symposium, NDSS 2014, The Internet Society, 2014.

[14] M. Naveed, M. Prabhakaran, and C. Gunter, "Dynamic searchable encryption via blind storage," IEEE Symposium on Security and Privacy, S&P 2014, pp.639–654, May 2014.

[15] I. Miers and P. Mohassel, "IO-DSSE: scaling dynamic searchable encryption to millions of indexes by improving locality," Network and Distributed System Security Symposium, NDSS 2017, 2017.

[16] T. Hirano, M. Hattori, Y. Kawai, N. Matsuda, M. Iwamoto, K. Ohta, Y. Sakai, and T. Munaka, "Simple, secure, and efficient searchable symmetric encryption with multiple encrypted indexes," Advances in Information and Computer Security, IWSEC 2016, ed. K. Ogawa and K. Yoshioka, Cham, pp.91–110, Springer International Publishing, 2016.

[17] K. Hayasaka, Y. Kawai, Y. Koseki, T. Hirano, K. Ohta, and M. Iwamoto, "Probabilistic generation of trapdoors: Reducing information leakage of searchable symmetric encryption," Cryptology and Network Security, CANS 2016, ed. S. Foresti and G. Persiano, Cham, pp.350–364, Springer International Publishing, 2016.

[18] K. Kurosawa, "Garbled searchable symmetric encryption," Financial Cryptography and Data Security, FC 2014, ed. N. Christin and R. Safavi-Naini, Lecture Notes in Computer Science, vol.8437, pp.234–251, Springer Berlin Heidelberg, 2014.

[19] K. Kurosawa, K. Sasaki, K. Ohta, and K. Yoneyama, "Uc-secure dynamic searchable symmetric encryption scheme," Advances in Information and Computer Security, IWSEC 2016, ed. K. Ogawa and K. Yoshioka, Cham, pp.73–90, Springer International Publishing, 2016.

[20] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans, "Efficient dynamic searchable encryption with forward privacy," Proceedings of Privacy Enhancing Technologies (PoPETs), vol.2018(1), pp.5–20, 2018.

[21] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, New York, NY, USA, pp.1465–1482, ACM, 2017.

[22] J. Ghareh Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, New York, NY, USA, pp.1038–1055, ACM, 2018.

[23] R. Bost, "$\Sigma o\varphi o\varsigma$: Forward secure searchable encryption," ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, New York, NY, USA, pp.1143–1154, ACM, 2016.

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

[25] The CALO Project, "Enron email dataset (may 7, 2015 version)." `https://www.cs.cmu.edu/~enron/`, 2015.

A Model in the CGKO papers

We describe a syntax of SSE defined in [2, 3].

Definition 8 (SSE [2, 3])**.** *An SSE scheme* Σ_{CGKO} *over* Λ *consists of five-tuple non-interactive algorithms* $\Sigma_{\mathsf{CGKO}} := (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Trpdr}, \mathsf{Srch}, \mathsf{Dec})$*, which are defined as follows:*

- $K \leftarrow \mathsf{Gen}(\kappa)$*: It is a probabilistic algorithm which takes a security parameter* κ *as input and outputs a secret key* K*.*
- $(\mathsf{I}, \mathbf{C}) \leftarrow \mathsf{Enc}(K, \mathbf{D})$*: It is a probabilistic algorithm which takes a security key* K *and* n *document files* $\mathbf{D} := (D_1, \ldots, D_n)$ *as input and outputs a secure index* I *and corresponding ciphertexts* $\mathbf{C} := (C_1, \ldots, C_n)$*.*
- $\tau_q \leftarrow \mathsf{Trpdr}(K, q)$*: It is a deterministic algorithm which takes a security key* K *and a keyword* $q \in \Lambda$ *as input and outputs a trapdoor* τ_q*.*
- $\mathcal{X}_q \leftarrow \mathsf{Srch}(\mathsf{I}, \tau_q)$*: It is a deterministic algorithm which takes a secure index* I *and a trapdoor* τ_q *for a keyword* q *and outputs a set* $\mathcal{X}_q$ *of identifiers as a search result.*
- $D_i \leftarrow \mathsf{Dec}(K, C_i)$*: It is a deterministic algorithm which takes a security key* K *and a ciphertext* C_i *as input and outputs its corresponding document file* D_i*.*

The above model requires the following search correctness: for all $\kappa \in \mathbb{N}$, for all $K \leftarrow \mathsf{Gen}(\kappa)$, for all possible $\mathbf{D}$, for all $(\mathsf{I}, \mathbf{C}) \leftarrow \mathsf{Enc}(K, \mathbf{D})$, and for all $q \in \Lambda$, we have

$$\begin{aligned}&(\mathsf{Srch}(\mathsf{I}, \mathsf{Trpdr}(K, q)) = \mathsf{ID}_q)\\ &\quad \wedge (D_i \leftarrow \mathsf{Dec}(K, C_i) \text{ for } \forall i \in [n]),\end{aligned}$$

with overwhelming probability.

When we ignore encryption and decryption procedures for documents (they can be realized independently of other algorithms by PCPA-secure SKE), the above syntax can be seen as a special case of ours (Def. 4). Indeed, we can construct an SSE scheme with our syntax from the above algorithms.

- $(k, \sigma^{(0)}, \mathsf{EDB}^{(0)}) \leftarrow \mathsf{Setup}(\kappa, \mathsf{DB})$: Run $K \leftarrow \mathsf{Gen}(\kappa)$ and $(\mathsf{I}, \mathbf{C}) \leftarrow \mathsf{Enc}(K, \mathbf{D})$, and return $k := K$, $\sigma^{(0)} := \varepsilon$, and $\mathsf{EDB}^{(0)} := \mathsf{I}$, where ε denotes an empty string. Note that we ignore $\mathbf{C}$ here.

Σ_{SSE2}: $\mathsf{Setup}(\kappa, \mathsf{DB}^{(0)})$

1: **parse** $\mathsf{DB}^{(0)} = \{(\mathsf{id}_i, \mathcal{W}_i)\}_{i=1}^{n}$
2: Set max from $\max\{|f_1|, \ldots, |f_n|\}$
3: **for** $\forall i \in [|\mathcal{W}|]$ **do**
4: **for** $\forall j \in [|\mathsf{ID}_{w_i}|]$ **do**
5: $\mathsf{Index}[\pi_k(w_i \| j)] := \mathsf{id}_{i,j}$ // $\mathsf{id}_{i,j}$: j-th id in ID_{w_i}
6: **for** $\forall j \in [n]$ **do**
7: **for** $\forall \beta \in [\mathsf{max} - |\mathcal{W}_{\mathsf{id}_j}|]$ **do**
8: $\mathsf{Index}[\pi_k(0^\lambda \| n + \beta)] := \mathsf{id}_j$ // add dummies
9: $\mathsf{EDB}^{(0)} := \mathsf{Index}$
10: **return** $(k, \sigma^{(0)} := \{n\}, \mathsf{EDB}^{(0)})$

Σ_{SSE2}: $\mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$

Client:
1: **for** $i = 1$ **to** n **do**
2: $\mathcal{T}_q^{(t)} \leftarrow \pi_k(q \| i)$ // $\mathcal{T}_q^{(t)}$: trapdoor
3: Send $\mathsf{trans}_1^{(t)} := \mathcal{T}_q^{(t)}$ to the server
Server:
4: **for** $\forall \mathtt{addr} \in \mathcal{T}_q^{(t)}$ **do**
5: **if** $\mathsf{Index}[\mathtt{addr}] \neq$ NULL **then**
6: $\mathcal{X}_q^{(t)} \leftarrow \mathsf{Index}[\mathtt{addr}]$ // $\mathcal{X}_q^{(t)}$: search result
7: Send $\mathsf{trans}_2^{(t)} := \mathcal{X}_q^{(t)}$ to the client
8: **return** $\mathsf{EDB}^{(t+1)} := \mathsf{Index}$
Client:
9: **return** $(\sigma^{(t+1)} := \sigma^{(t)}, \mathcal{X}_q^{(t)})$

Figure 16: The original SSE-2 scheme.

- $(\sigma^{(t+1)}, \mathcal{X}_q^{(t)}; \mathsf{EDB}^{(t+1)}) \leftarrow \mathsf{Search}(k, q, \sigma^{(t)}; \mathsf{EDB}^{(t)})$:

  $(\sigma^{(t+1)}, \mathcal{X}_q^{(t)}) \leftarrow \mathsf{Search_c}(k, q, \sigma^{(t)})$. Run $\tau_q \leftarrow \mathsf{Trpdr}(K, q)$, and send τ_q as $\mathsf{trans}_1^{(t)}$ to the server. Receiving $\mathsf{trans}_2^{(t)}$, return $\mathcal{X}_q^{(t)} := \mathsf{trans}_2^{(t)}$ and $\sigma^{(t+1)} := \varepsilon$.

  $\mathsf{EDB}^{(t+1)} \leftarrow \mathsf{Search_s}(\mathsf{EDB}^{(t)})$. Receiving $\mathsf{trans}_1^{(t)}$, run $\mathsf{Srch}(\mathsf{I}, \mathsf{trans}_1^{(t)})$ and sends the output as $\mathsf{trans}_2^{(t)}$ to the client. Return $\mathsf{EDB}^{(t+1)} := \mathsf{EDB}^{(t)}$ $(= \mathsf{I})$.

B The Original SSE-2 Scheme

We describe the original SSE-2 scheme $\Sigma_{\mathsf{SSE2}} = (\mathsf{Setup}, \mathsf{Search})$ in Fig. 16. Curtmola et al. actually described the original SSE-2 scheme as if it is an inverted-index-based scheme, although the basic idea behind SSE-2 is the forward index as described in Section 3.

As already mentioned in [8], Curtmola et al.'s dummy addition procedure had fateful flaws. More specifically, in line 8 of Setup, dummy entries are added to $\mathsf{Index}[\pi_k(0^\lambda \| n+1)], \ldots, \mathsf{Index}[\pi_k(0^\lambda \| n + \mathsf{max} - |\mathcal{W}_i|)]$ for every file $f_i \in \mathsf{DB}^{(0)}$. Namely, the dummies are overwritten for every $i \in [n]$, and hence the SSE-2 construction does not work well.

C Security Proof of Our Dynamic SSE Scheme

First, we show that S can simulate all transcripts during the execution of Update by using $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{op}, \mathsf{in})$. In the following, we suppose that an identity id is never reused again once the corresponding file f_{id} is deleted.[14]

For query $= (\mathsf{upd}, \mathsf{add}, (\mathsf{id}, \mathcal{W}_{\mathsf{id}}))$: In $\mathsf{Real}_{\mathsf{D}}(\kappa, Q)$, Update is executed, and the corresponding transcript $\mathsf{trans}_1^{(t)} = (\mathsf{id}, \mathcal{U}_{\mathsf{id}}^{(t)})$ consists of id and $\mathsf{max}_{\mathsf{id}}$ random strings, which are used as addresses

[14] This handling for identifiers is highly recommended in practice since the server notices that previously-deleted files are re-added if the identifiers are reused. Moreover, though we can also prove this theorem in the setting where id is reused, it seems to contradict forward privacy, which guarantees that the addition procedure leaks no information on unique keywords contained in newly-added files, since the server can notice the previous search results related to id at the point when id is re-added.

for id in $\mathsf{EDB}^{(t+1)}$. Therefore, roughly speaking, if the simulator S randomly chooses $\mathsf{max}_{\mathsf{id}}$ *unused* $(\lambda+\ell+1)$-bit strings as addresses for id in $\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}(\kappa, Q)$, D cannot distinguish the two experiments due to the security of π (see Def. 1).

Formally, S simulates the transcript $\mathsf{trans}_1^{(t)} = (\mathsf{id}, \mathcal{U}_{\mathsf{id}}^{(t)})$ as follows. Due to the addition procedure, addresses of previously-registered files definitely store the corresponding identities. On the other hand, due to the search procedure, previous trapdoors for a certain keyword q are empty addresses if the corresponding files do not contain q. Therefore, due to the search correctness, it is not appropriate to just choose empty addresses of Index, which are unused strings as addresses of files ever to be registered, as addresses for id. Hence, addresses used for $\mathcal{U}_{\mathsf{id}}^{(t)}$ have to be all fresh, i.e., S has to choose addresses that have not been used as either addresses nor trapdoors for previously-registered files.

To capture this, we additionally define the following notations. Let $\mathsf{List}_{\mathsf{id}}$ be a list of all pairs of an identifier and a global counter when it was registered, i.e., $\mathsf{List}_{\mathsf{id}} = \{(\mathsf{id}, t')\}$, and $\mathcal{I}^{(t)}$ be a set of identifiers stored in the database at t. Let $\mathsf{List}_{\mathsf{addr}} := \bigcup_{(\mathsf{id}', t') \in \mathsf{List}_{\mathsf{id}}} \mathcal{U}_{\mathsf{id}'}^{(t')}$ be a list of all addresses that have been registered at least once by t. Note that $\mathsf{List}_{\mathsf{addr}}$ might include addresses deleted by t. Let $\mathsf{List}_{\mathsf{used}}$ be a list of all addresses that have been used for the response of search queries (i.e., used as trapdoors) at least once by t.

For $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{add}, (\mathsf{id}, \mathcal{W}_{\mathsf{id}})) = (\mathsf{id}, |f_{\mathsf{id}}|)$, S computes $\mathsf{max}_{\mathsf{id}}$ from $|f_{\mathsf{id}}|$ and Λ, and randomly chooses $\mathsf{max}_{\mathsf{id}}$ *unused* addresses. Namely, S repeats the following procedure $\mathsf{max}_{\mathsf{id}}$ times:

1. $\mathtt{addr} \xleftarrow{\$} \{0,1\}^{\lambda+\ell+1} \setminus (\mathsf{List}_{\mathsf{addr}} \cup \mathsf{List}_{\mathsf{used}})$.
2. $\mathsf{List}_{\mathsf{addr}} \leftarrow \mathtt{addr}$.
3. $\mathcal{U}_{\mathsf{id}}^{(t)} \leftarrow \mathtt{addr}$.

All addresses in $\mathsf{List}_{\mathsf{addr}}$ are distinct from each other since π is a permutation, and look random due to the security of π. Finally, S adds (id, t) to $\mathsf{List}_{\mathsf{id}}$, and sets $\mathcal{I}^{(t)} := \mathcal{I}^{(t-1)} \cup \{\mathsf{id}\}$. Hence, S can simulate $\mathsf{Update}(k, \mathsf{add}, (\mathsf{id}, \mathcal{W}_{\mathsf{id}}), \sigma^{(t)}; \mathsf{EDB}^{(t)})$ by only using $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{add}, (\mathsf{id}, \mathcal{W}_{\mathsf{id}}))$.

For $\mathsf{query} = (\mathtt{upd}, \mathsf{del}, \mathsf{id})$: In $\mathsf{Real}_{\mathsf{D}}(\kappa, Q)$, the client sends the transcript $\mathsf{trans}_1^{(t)} := \mathsf{id}$, and the server deletes the corresponding addresses that store id. It is obvious that it can be easily simulated by S using $\mathcal{L}_{\mathsf{Upd}}(t, \mathsf{del}, \mathsf{id}) = \mathsf{id}$. S sets $\mathcal{I}^{(t)} := \mathcal{I}^{(t-1)} \setminus \{\mathsf{id}\}$.

For $\mathsf{query} = (\mathtt{srch}, q)$: In $\mathsf{Real}_{\mathsf{D}}(\kappa, Q)$, the client first sends a request as $\mathsf{trans}_1^{(t)}$ and receives $\mathsf{trans}_2^{(t)} = \mathcal{I}$ back, where $\mathcal{I}$ is exactly the same as $\mathcal{I}^{(t-1)}$ that S maintains. Then, the client computes $\pi_k(0\|q\|\mathsf{id})$ for all $\mathsf{id} \in \mathcal{I}$, and sends the server $\mathsf{trans}_3^{(t)} = \mathcal{T}_q^{(t)} := \{\pi_k(0\|q\|\mathsf{id}) \mid \mathsf{id} \in \mathcal{I}\}$ as *trapdoors*. From the correctness, it holds $\mathsf{Index}[\pi_k(0\|q\|\mathsf{id})] = \mathsf{id}$ if f_{id} contains q; it holds $\mathsf{Index}[\pi_k(0\|q\|\mathsf{id})] = \mathrm{NULL}$ otherwise. Moreover, it holds $\mathcal{X}_q^{(t)} = \mathsf{ID}_q^{(t)}$.

We construct S that simulates the above procedure correctly as follows. First of all, S sets $\mathcal{I}^{(t)} := \mathcal{I}^{(t-1)}$. Then, we have to consider two cases depending on $\mathcal{L}_{\mathsf{Srch}}(t, q) = (\mathsf{SP}_q^{(t)}, \mathsf{AP}_q^{(t)})$:

(1) It is the first time to search for q, i.e., $\mathsf{SP}_q^{(t)} = \{t\}$.

(2) q has been queried before, i.e., $\mathsf{SP}_q^{(t)} \neq \{t\}$.

The reason why we consider the two cases is that trapdoors at the first search for a keyword q should be chosen at random, but those at at two and subsequent searches should be the same as the first search. We give an illustrative diagram of the simulation for Search in Fig. 17.

(1) *It is the first time to search for q, i.e.,* $\mathsf{SP}_q^{(t)} = \{t\}$. In $\mathsf{Ideal}_{\mathsf{D},\mathsf{S},\mathcal{L}}(\kappa, Q)$, S simulates the above real procedures by inverse process. Note that S knows $\mathcal{L}_{\mathsf{Srch}}(t, q) = (\mathsf{SP}_q^{(t)}, \mathsf{AP}_q^{(t)})$. S randomly

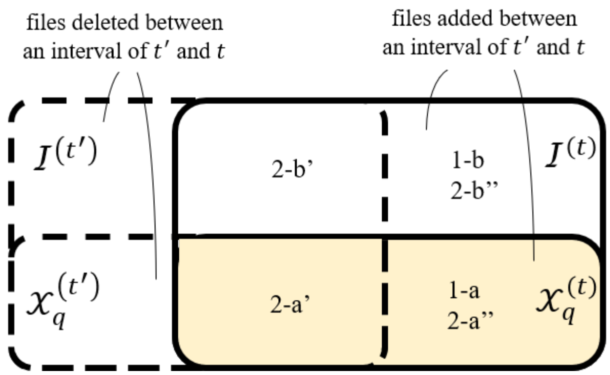

Figure 17: An illustrative diagram of the search simulation, where $t' := \max \mathsf{SP}_q^{(t)} \setminus \{t\}$. S simulates trapdoors for all $\mathsf{id} \in \mathcal{I}^{(t)}$. Note that in the case of (1), we have $\mathsf{SP}_q^{(t)} \setminus \{t\} = \emptyset$.

chooses an unused address $\mathtt{addr}_{\mathsf{id},q}$ as a trapdoor for q and id for all $\mathsf{id} \in \mathcal{I}$, but the domain from which $\mathtt{addr}_{\mathsf{id},q}$ is chosen depends on whether $\mathsf{id} \in \mathsf{AP}_q^{(t)}$ or not.

(1-a) Every identity $\mathsf{id} \in \mathsf{AP}_q^{(t)}$ should be stored at $\mathtt{addr}_{\mathsf{id},q}$. Note that S needs to avoid choosing addresses already used as trapdoors for other keywords contained in f_{id}. Therefore, S chooses $\mathtt{addr}_{\mathsf{id},q}$ from $\mathcal{U}_{id}^{(t')} \setminus \mathsf{List}_{\mathsf{used}}$, where t' is a counter such that $(\mathsf{id}, t') \in \mathsf{List}_{\mathsf{id}}$.

(1-b) For every $\mathsf{id} \in \mathcal{I}^{(t)} \setminus \mathsf{AP}_q^{(t)}$, the corresponding trapdoor should be an empty address. However, we have to pay attention to the fact that some empty addresses (i.e., addresses stored in $\mathsf{List}_{\mathsf{used}}$) were assigned to trapdoors for other previously-searched keywords. Therefore, S randomly chooses $\mathtt{addr}_{\mathsf{id},q}$ from $\{0,1\}^{\lambda+\ell+1} \setminus (\mathsf{List}_{\mathsf{used}} \cup \mathsf{List}_{\mathsf{addr}})$.

S then adds $\mathtt{addr}_{\mathsf{id},q}$ to each of $\mathcal{T}_q^{(t)}$ and $\mathsf{List}_{\mathsf{used}}$. Therefore, S can simulate the search procedure by setting $\mathsf{trans}_1^{(t)} := \mathtt{request}$, $\mathsf{trans}_2^{(t)} := \mathcal{I}^{(t)}$, $\mathsf{trans}_3^{(t)} := \mathcal{T}_q^{(t)}$ and $\mathsf{trans}_4^{(t)} := \mathsf{AP}_q^{(t)}$. S maintains a list $\mathsf{SrchList}_q^{(t)} := \{(\mathtt{addr}_{\mathsf{id},q}, \mathsf{id}) \mid \mathsf{id} \in \mathcal{I}^{(t)}\}$, which maintains all pairs of a trapdoor for id and q and the corresponding identity id at t, for the case (2).

(2) *q has been queried before, i.e.,* $\mathsf{SP}_q^{(t)} \neq \{t\}$. In this case, S basically follows the same procedure as the case (1). Therefore, S simulates trapdoors for the following two kinds of identifiers:

(2-a) id that appears in the search result, i.e., $\mathsf{id} \in \mathsf{AP}_q^{(t)}$.

(2-b) id that does not appear in the search result but that is stored in the current database, i.e., $\mathsf{id} \in \mathcal{I}^{(t)} \setminus \mathsf{AP}_q^{(t)}$.

For the case (2-a), unlike the case (1-a), we have to care about the fact that $\mathsf{AP}_q^{(t)}$ contains two kinds of identifiers:

(2-a') id that already appeared in the last search result, i.e., $\mathsf{id} \in \mathsf{AP}_q^{(t')} \cap \mathsf{AP}_q^{(t)}$, where $t' := \max(\mathsf{SP}_q^{(t)} \setminus \{t\})$.[15]

[15] If we allow the client to reuse id, (i.e., the same id is assigned to a deleted file for addition), we have to consider all $t'' \in \mathsf{SP}_q^{(t)} \setminus \{t\}$, not just $t' := \max(\mathsf{SP}_q^{(t)} \setminus \{t\})$. The reason for this is that there might exist id that appears in the search results at t and t'', but does not appear in the result at t', where $\mathsf{SP}_w^{(t)} := \{t, t', t''\}$ and $t > t' > t''$. Thus, in such a case, we should consider $\mathsf{id} \in \bigcup_{t'' \in \mathsf{SP}_q^{(t)} \setminus \{t\}} \mathsf{AP}_q^{(t'')}$.

(2-a") id added to Index after the last search for q, i.e., $\mathsf{id} \in \mathsf{AP}_q^{(t)} \setminus \mathsf{AP}_q^{(t')}$.

Note that we have $(\mathsf{AP}_q^{(t')} \cap \mathsf{AP}_q^{(t)}) \cup (\mathsf{AP}_q^{(t)} \setminus \mathsf{AP}_q^{(t')}) = \mathsf{AP}_q^{(t)}$ since it holds $(\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{B} \setminus \mathcal{A}) = \mathcal{B}$ for any finite sets $\mathcal{A}$ and $\mathcal{B}$. For the case (2-a'), S has to use the same trapdoors as previously-used ones. It can be done easily; for $\mathsf{id} \in \mathsf{AP}_q^{(t')} \cap \mathsf{AP}_q^{(t)}$, S retrieves $(\mathtt{addr}_{\mathsf{id},q}, \mathsf{id})$ from $\mathsf{SrchList}_q^{(t')}$, and adds $\mathtt{addr}_{\mathsf{id},q}$ to $\mathcal{T}_q^{(t)}$. For the case (2-a"), S generates trapdoors as in the case (1-a), and adds them to each of $\mathcal{T}_q^{(t)}$ and $\mathsf{List}_{\mathsf{used}}$.

Similarly, for the case (2-b), we need to divide into the following two cases:

(2-b') id that does not contain q and that already existed in the database at the the last search for q, i.e., $\mathsf{id} \in (\mathcal{I}^{(t')} \setminus \mathsf{AP}_q^{(t')}) \cap (\mathcal{I}^{(t)} \setminus \mathsf{AP}_q^{(t)})$.

(2-b") id that does not contain q and that was added to Index after the last search for q, i.e., $\mathsf{id} \in (\mathcal{I}^{(t)} \setminus \mathsf{AP}_q^{(t)}) \setminus (\mathcal{I}^{(t')} \setminus \mathsf{AP}_q^{(t')})$.

Note that we have $((\mathcal{I}^{(t')} \setminus \mathsf{AP}_q^{(t')}) \cap (\mathcal{I}^{(t)} \setminus \mathsf{AP}_q^{(t)})) \cup ((\mathcal{I}^{(t)} \setminus \mathsf{AP}_q^{(t)}) \setminus (\mathcal{I}^{(t')} \setminus \mathsf{AP}_q^{(t')})) = \mathcal{I}^{(t)} \setminus \mathsf{AP}_q^{(t)}$, as in the case (2-a). For the case (2-b'), S has to use the same trapdoors as previously-used ones. Namely, for $\mathsf{id} \in (\mathcal{I}^{(t')} \setminus \mathsf{AP}_q^{(t')}) \cap (\mathcal{I}^{(t)} \setminus \mathsf{AP}_q^{(t)})$, S retrieves $(\mathtt{addr}_{\mathsf{id},q}, \mathsf{id})$ from $\mathsf{SrchList}_q^{(t')}$, and adds $\mathtt{addr}_{\mathsf{id},q}$ to $\mathcal{T}_q^{(t)}$. For the case (2-b"), S generates trapdoors as in the case (1-b), and adds them to each of $\mathcal{T}_q^{(t)}$ and $\mathsf{List}_{\mathsf{used}}$.

S finally sets $\mathsf{SrchList}_q^{(t)} := \{(\mathtt{addr}_{\mathsf{id},q}, \mathsf{id}) \mid \mathsf{id} \in \mathcal{I}^{(t)}\}$. Thus, S can correctly simulate the search procedure by setting $\mathsf{trans}_1^{(t)} := \mathtt{request}$, $\mathsf{trans}_2^{(t)} := \mathcal{I}^{(t)}$, $\mathsf{trans}_3^{(t)} := \mathcal{T}_q^{(t)}$, and $\mathsf{trans}_4^{(t)} := \mathsf{AP}_q^{(t)}$. $\square$

D Operation Example of SSE-1

We describe a small example of SSE-1 below.

Setting: Dictionary $\Lambda = \{w_1, w_2, \ldots, w_6\}$, where w_1 is 1 bits, w_2, w_3 are 2 bits, and w_4, w_5, w_6 are 3 bits. The stored files are $f_1 = (\mathsf{id}_1, \mathcal{W}_1 = \{w_2, w_4\})$, $f_2 = (\mathsf{id}_2, \mathcal{W}_2 = \{w_1, w_2\})$, and $f_3 = (\mathsf{id}_3, \mathcal{W}_2 = \{w_1, w_2, w_3\})$.

In this setting, $\mathsf{max}_1 = 3$ since $|w_1| + |w_2| + |w_3| \le |f_1| < |w_1| + |w_2| + |w_3| + |w_4|$. Similarly, $\mathsf{max}_2 = 2$ and $\mathsf{max}_3 = 3$. Thus, $\mathsf{max}_{\mathsf{DB}} = 8$, and the number of dummy nodes is one since $\mathsf{max}_{\mathsf{DB}} - N = 1$

Setup: In lines 3–14 of the setup procedure, following seven nodes are created in Array.

- $\mathsf{Array}[\psi_{K_1}(1)] := \mathsf{E}(k_{w_1,1}, \mathsf{N}_{w_1,1})$
  where $\mathsf{N}_{w_1,1} := \mathsf{id}_2 \| \psi_{K_1}(2) \| k_{w_1,2}$
- $\mathsf{Array}[\psi_{K_1}(2)] \leftarrow \mathsf{E}(k_{w_1,2}, \mathsf{N}_{w_1,2})$
  where $\mathsf{N}_{w_1,2} := \mathsf{id}_3 \| 0^{s+\kappa}$
- $\mathsf{Array}[\psi_{K_1}(3)] := \mathsf{E}(k_{w_2,1}, \mathsf{N}_{w_2,1})$
  where $\mathsf{N}_{w_2,1} := \mathsf{id}_1 \| \psi_{K_1}(4) \| k_{w_2,2}$
- $\mathsf{Array}[\psi_{K_1}(4)] := \mathsf{E}(k_{w_2,2}, \mathsf{N}_{w_2,2})$
  where $\mathsf{N}_{w_2,2} := \mathsf{id}_2 \| \psi_{K_1}(5) \| k_{w_2,3}$
- $\mathsf{Array}[\psi_{K_1}(5)] := \mathsf{E}(k_{w_2,3}, \mathsf{N}_{w_2,3})$
  where $\mathsf{N}_{w_2,3} := \mathsf{id}_3 \| 0^{s+\kappa}$
- $\mathsf{Array}[\psi_{K_1}(6)] := \mathsf{E}(k_{w_3,1}, \mathsf{N}_{w_3,1})$
  where $\mathsf{N}_{w_3,1} := \mathsf{id}_3 \| 0^{s+\kappa}$

Row No.	Value
$\pi_{K_3}(w_1)$	$(\psi_{K_1}(1) \Vert k_{w_1,1}) \oplus f_{K_2}(w_1)$
$\pi_{K_3}(w_2)$	$(\psi_{K_1}(3) \Vert k_{w_2,1}) \oplus f_{K_2}(w_2)$
$\pi_{K_3}(w_3)$	$(\psi_{K_1}(6) \Vert k_{w_3,1}) \oplus f_{K_2}(w_3)$
$\pi_{K_3}(w_4)$	$(\psi_{K_1}(7) \Vert k_{w_4,1}) \oplus f_{K_2}(w_3)$
$v_{w_5} \xleftarrow{\$} \{0,1\}^v \setminus \Pi_{\mathcal{W}}$	$c_{w_5} \xleftarrow{\$} \{0,1\}^{s+\kappa}$
$v_{w_6} \xleftarrow{\$} \{0,1\}^v \setminus \Pi_{\mathcal{W}}$	$c_{w_6} \xleftarrow{\$} \{0,1\}^{s+\kappa}$

Figure 18: An Address Table (Example)

- $\mathsf{Array}[\psi_{K_1}(7)] \coloneqq \mathsf{E}(k_{w_4,1}, \mathsf{N}_{w_4,1})$
  where $\mathsf{N}_{w_4,1} \coloneqq \mathsf{id}_1 \Vert 0^{s+\kappa}$

Furthermore, at line 16, the following node is created as a dummy node.

- $\mathsf{Array}[\psi_{K_1}(8)] \xleftarrow{\$} \{0,1\}^{l+s+\kappa}$

Also, Table is described in Fig. 18.

Search: Consider the case of searching w_2. The client first sends $(\pi_{K_3}(w_2), f_{K_2}(w_2))$ as the trapdoor to the server. The server obtains $\psi_{K_1}(3)$ and $k_{w_2,1}$ by $\mathsf{Table}[\pi_{K_3}(w_2)] \oplus f_{K_2}(w_2)$. The server initiates the sequential decryption using $\psi_{K_1}(3)$ and $k_{w_2,1}$ as follows.

1. The server obtains $\mathsf{N}_{w_2,1} = \mathsf{id}_1 \Vert \psi_{K_1}(4) \Vert k_{w_2,2}$ by decrypting $\mathsf{Array}[\psi_{K_1}(3)]$ using $k_{w_2,1}$
2. The server obtains $\mathsf{N}_{w_2,2} = \mathsf{id}_2 \Vert \psi_{K_1}(5) \Vert k_{w_2,3}$ by decrypting $\mathsf{Array}[\psi_{K_1}(4)]$ using $k_{w_2,2}$
3. The server obtains $\mathsf{N}_{w_2,3} = \mathsf{id}_3 \Vert 0^{s+\kappa}$ by decrypting $\mathsf{Array}[\psi_{K_1}(5)]$ using $k_{w_2,3}$, and it terminates the sequential decryption.

As a result, the server gets the result $\mathsf{ID}_{w_2} = \{\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3\}$, which is correct.