This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3288160

Authorized Keyword Search on Mobile Devices in Secure Data Outsourcing

Qin Liu, *Member, IEEE,* Yu Peng, *Graduate Student Member, IEEE,* Hongbo Jiang, *Senior Member, IEEE,* Jie Wu, *Fellow, IEEE,* Tian Wang, *Member, IEEE,* Tao Peng, *Member, IEEE,* and Guojun Wang, *Member, IEEE*

Abstract—With the increasing awareness of secure data outsourcing, dynamic searchable symmetric encryption (DSSE) that enables searches and updates over encrypted data has begun to receive growing attention. Despite promising, existing DSSE schemes with forward and backward privacy are still hard to achieve authorized keyword searches on mobile devices while supporting secure and flexible updates. In this paper, we propose a DSSE scheme, named FLY++ based on a flexible index structure Hybrid that incorporates the merits of inverted indexes and forward indexes while compacting the index size. Specifically, FLY++ encrypts the newly added data with a fresh key and disperses previous keys into Hybrid for forward privacy, while applying symmetric puncturable encryption (SPE) and a dual-key mechanism to realize backward privacy further. Compared with the state-of-the-art work, FLY++ has the following advantages: (1) *Authorized search.* It dispenses with caching or re-encrypting search results, enabling a mobile device to search only designated keywords over the data outsourced before authorization. (2) *Flexibility.* It not only allows for sublinear search time, but also simultaneously supports fine-grained and coarse-grained updates of outsourced data. The detailed security analysis and extensive experiments conducted on a real dataset demonstrate the security and practicality of FLY++, respectively.

Index Terms—Data outsourcing, dynamic searchable symmetric encryption, forward privacy, backward privacy, authorized search.

1 Introduction

Cloud computing that centralizes enormous computing and storage resources provides various services in a pay-per-use fashion, such as data analysis, gesture recognition and so on [1], [2]. As ever-growing amounts of consumers outsource data for lower costs and better performance, users' sensitive data is at heightened risk of disclosure and abuse. For example, recent news about Akamai servers leaking about 20GB of Intel confidential documents and Amazon Simple Storage Service (S3) leaking personal medical data [3]. To guarantee data confidentiality, existing studies [4], [5] suggest encrypting data before outsourcing.

Dynamic searchable symmetric encryption (DSSE) [6]–[11] that enables keyword-based searches and continuous updates over encrypted data has become a promising solution for secure data outsourcing. Compared with static SSE [12], [13], DSSE is typically equipped with *forward privacy* requiring the newly added data to be unsearchable by previous search tokens [14]–[21], and *backward privacy* requiring the deleted data to be inaccessible by subsequent search tokens [22]–[28]. Even so, *how to achieve authorized keyword searches over dynamic outsourced data in a secure, flexible, and efficient way* is still an open problem. Let us consider the application scenario as shown in Fig. 1. Hospital A deploys medical Internet of Things (IoT) devices for continuous data gathering, and periodically uploads the incrementally collected data to Amazon S3, so that the authorized mobile devices can retrieve data of interest anytime and anywhere. The medical data is relevant to patient privacy, and will be desensitized and encrypted before outsourcing. For security reasons, each mobile device is restricted to search a subset of keywords over the data already generated, and needs to be reauthorized if it wants to access the data newly gathered. Moveover, hospital A may need to remove/modify the sensitive data that is accidentally uploaded to Amazon S3.

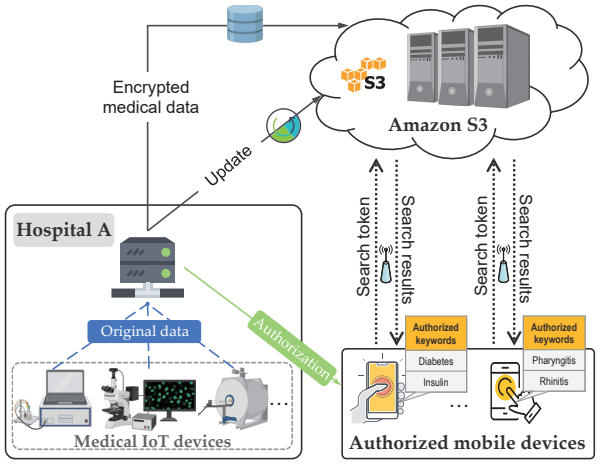


Fig. 1. Application scenario. Hospital A outsources the medical data to Amazon S3 for cost saving and ubiquitous access.

- *Qin Liu, Yu Peng and Hongbo Jiang are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan Province, P. R. China, 410082. E-mail: gracelq628@hnu.edu.cn; pengyu411@hnu.edu.cn; hongbojiang2004@gmail.com*
- *Jie Wu is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA. E-mail: jiewu@temple.edu*
- *Tian Wang is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University & UIC, Zhuhai, Guangdong Province, P. R. China, 519000. E-mail: cs_tianwang@163.com*
- *Tao Peng and Guojun Wang are with the School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou, Guangdong Province, P. R. China, 510006. E-mail: pengtao@gzhu.edu.cn; csgjwang@gzhu.edu.cn*

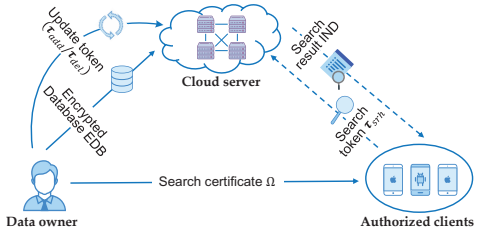


Fig. 2. System model. The communication channels between DO and AC are assumed to be secured under SSL/TLS.

In this scenario, the adopted DSSE scheme should meet the following requirements: (1) *Authorized search*. A mobile device can only search designated keywords over the data outsourced before authorization. (2) *Lightweight*. A mobile device with limited resources can efficiently retrieve the desired data through authorized keywords. (3) *Flexibility*. Hospital A can update (modify, add and delete) the outsourced data on demand. (4) *Secure deletion.* Once the sensitive data is deleted before being accessed, the cloud server cannot infer any useful information anymore. Unfortunately, existing DSSE schemes only partially address these requirements due to the following reasons: First, most solutions let cloud servers cache the results of previous search queries in plaintext and use them to answer later queries. This avoids heavy re-encryption operations, but leaves the data owner to rely on cloud servers to implement authorized search. Second, existing solutions support only fine-grained updates (i.e., adding/deleting a specific keyword in a file at a time) or only coarse-grained updates (i.e., adding/deleting all the keywords in a file at once), and thus lack flexibility.

To satisfy all the above properties, we propose a DSSE scheme, named FLY++, which encrypts the newly added data with a fresh key while utilizing symmetric puncturable encryption (SPE) [28] to achieve both forward and backward privacy. Specifically, FLY++ is built based on a flexible index Hybrid that incorporates the merits of forward and inverted indexes, not only achieving sublinear search time, but also simultaneously supporting fine-grained and coarse-grained updates. As for authorized search and lightweight, FLY++ disperses previous keys into Hybrid, and adopts a dual-key mechanism to avoid caching and re-encrypting search results, so that a mobile device with a small-sized certificate can rapidly generate search tokens for preassigned keywords. Finally, the backward-private (BP) property of FLY++ enables the deleted data to be unsearchable, and thus naturally guarantees secure deletion. To better understand the mechanism of each adopted technique, we construct FLY++ incrementally: We first provide FLY, a forward-private (FP) DSSE scheme to realize authorized searches and flexible updates on mobile devices, on this basis, we then propose FLY+ that further achieves backward privacy but lacks flexibility, and finally we obtain FLY++, the full-featured scheme. Our contributions are summarized as follows:

- To the best of our knowledge, this is the first attempt to devise a DSSE scheme with forward and backward privacy to simultaneously achieve authorized search, lightweight, flexible update, and secure deletion.
- We design an index, Hybrid, to support fast searches and flexible updates on encrypted outsourced data.
- We propose FLY, FLY+, and FLY++ with a trade-off between efficiency and security, which allow cloud servers to store ciphertexts without re-encryption.
- We formally analyze the security of all the proposed schemes, and evaluate their performance on a real dataset, Enron Email. The results show that the proposed schemes are extremely efficient in the search phase and outperform the state-of-the-art work.

TABLE 1
Summary of Notations

Notations	Descriptions
w, f_i	A keyword and a file identifier
$\mathsf{DB}(f_i)$	A set of keywords contained in file f_i
$\mathsf{DB}, \mathsf{EDB}$	A database $(f_i, \mathsf{DB}(f_i))_{i=1}^n$ and an encrypted database
W	The set of keywords in DB (i.e., $\mathsf{W} = \cup_{i=1}^n \mathsf{DB}(f_i)$)
$\mathsf{DB}(w)$	The set of files containing keyword w

Paper Organization. We formulate the problem in Section 2, and construct schemes in Sections 3 and 4. After giving the security analyses in Section 5, we evaluate our schemes in Section 6. Finally, we introduce the related work in Section 7 before concluding the paper in Section 8.

2 Problem Formulation

2.1 System and Threat Models

Corresponding to the scenario shown in Fig. 1, our system model consists of three types of entities as shown in Fig. 2:

• **Data Owner (DO).** Hospital A as the DO possesses a large-scale database DB, and outsources the searchable encrypted database EDB for cost-saving and ubiquitous access. The DO is allowed to update EDB as required and grant authorized clients different search permission.

• **Cloud Server (CS).** Amazon S3 as the CS pools massive resources to offer data storage and query services. Given an update token (τ_{add} or τ_{del}) sent by the DO, the CS can update EDB appropriately. Upon receiving a search token τ_{srh} from the authorized client, the CS evaluates τ_{srh} on EDB and returns the search result IND.

• **Authorized Client (AC).** A mobile device as the AC obtains a search certificate Ω from the DO in advance. With Ω, the AC can generate search tokens for authorized keywords and search the data outsourced before authorization.

In our threat model, the DO is assumed to be trustworthy, and the CS and the AC are assumed to be honest-but-curious attackers [29]. That is to say, the attackers will faithfully execute the predefined protocols, but may try to exploit additional information from the encrypted database and search/update tokens. For example, the AC may want to access the newly generated data outside its permission, and the CS may try to determine whether a new/deleted file containing a specific keyword or not, violating forward/backward privacy. Furthermore, the CS and the AC belonging to different parties are assumed to not collude.

2.2 Notations

Let $\lambda \in \mathbb{N}$ be a security parameter and let notation $\mathbf{0}$ represent a string of 0s with length λ. For $x \in \mathbb{N}$, notation $[x]$ is used to denote the set of integers $\{1, \ldots, x\}$. We denote the set of binary strings of length x by $\{0,1\}^x$ and the set of finite binary strings by $\{0,1\}^*$. Notation $\|$ denotes string concatenation. For a finite set X, notation $|X|$ denotes its cardinality, and $(x_1, \ldots, x_k) \xleftarrow{\$} X$ means that x_i is sampled uniformly from X for $i \in [k]$. For quick reference, the most relevant notations for our schemes are shown in Table 1.

Algorithm 1 SPE

$\underline{\mathsf{GenKey}(\mathsf{lks}^{(w)}, \tau) \rightarrow (\mathsf{key}^{(\tau)})}$

$sk_0 \leftarrow \mathsf{lks}^{(w)}$
for $i \in [d]$ **do**
  $sk_i = H(sk_{i-1})$
$\mathsf{key}^{(\tau)} = \oplus_{i=0}^{d} \widetilde{F}(sk_i, \tau)$

$\underline{\mathsf{IncPun}(\mathsf{lks}^{(w)}, \tau) \rightarrow (\mathsf{lks}'^{(w)}, \mathsf{pks}^{(\tau)})}$

$\mathsf{lks}'^{(w)} \leftarrow H(\mathsf{lks}^{(w)})$
$\mathsf{pks}^{(\tau)} \leftarrow \widetilde{\mathsf{F}}.\mathsf{Punc}(\mathsf{lks}^{(w)}, \tau)$

$\underline{\mathsf{RegenKey}(\mathsf{SK}, \tau) \rightarrow (\mathsf{key}^{(\tau)})}$

Parse SK as $(\mathsf{lks}^{(w)}, \mathrm{PKS})$; $t \leftarrow |\mathrm{PKS}|$
Parse PKS as $\{\mathsf{pks}^{(\tau_1)}, \ldots, \mathsf{pks}^{(\tau_t)}\}$; $sk_t \leftarrow \mathsf{lks}^{(w)}$
if $t < d$ **then**
  for $i = t+1$ to d **do**
    $sk_i = H(sk_{i-1})$
$\mathsf{key}^{(\tau)} = \bigoplus_{i=1}^{t} \widetilde{\mathsf{F}}.\mathsf{Eval}(\mathsf{pks}^{(\tau_i)}, \tau) \oplus \bigoplus_{i=t}^{d} \widetilde{F}(sk_i, \tau)$

2.3 Puncturable Pseudorandom Function (PPRF)

A PPRF [30] can be regarded as a PRF $\widetilde{F} : \{0,1\}^\lambda \times X \rightarrow Y$ together with a pair of algorithms $\widetilde{\mathsf{F}}.\mathsf{Punc}$ and $\widetilde{\mathsf{F}}.\mathsf{Eval}$:

- $\widetilde{\mathsf{F}}.\mathsf{Punc}$ is a probabilistic algorithm that takes the key $k \in \{0,1\}^\lambda$ and a set of punctured elements $S \subseteq X$ as input, and outputs the punctured key k_S.
- $\widetilde{\mathsf{F}}.\mathsf{Eval}$ is a deterministic algorithm taking the punctured key k_S and an element $x' \in X$ as input. It outputs $\widetilde{F}(k, x')$ if and only if $x' \notin S$ (i.e., x' hasn't been punctured).

In brief, given the input domain X, PPRF allows the holder of a secret key k to punctures a set of elements $S \subseteq X$, rendering any element in S impossible to be mapped into the output domain Y. Moreover, the holder is allowed to puncture either a single element ($|S| = 1$) at a time or a batch of elements ($|S| > 1$) at once.

2.4 Symmetric Puncturable Encryption (SPE)

SPE [28] is built based on PPRFs. In a nutshell, SPE associates each keyword/file pair with a tag, and allows a key holder to delete a fixed number of pairs by puncturing their tags. The original SPE mainly consists of three polynomial-time algorithms $\mathsf{SPE} = (\mathsf{Enc}, \mathsf{IncPun}, \mathsf{Dec})$, where algorithm Enc (resp. Dec) consists of two steps: secret key generation (resp. regeneration) and symmetric encryption (resp. decryption). Instead of using symmetric encryption (SE), we employ the dual-key mechanism to encrypt keyword/file pairs, and thus we remove the symmetric encryption (resp. decryption) step from algorithm Enc (resp. Dec). Let $H : \{0,1\}^\lambda \rightarrow \{0,1\}^\lambda$ be a cryptographic hash function, and let $\widetilde{F} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be a PPRF. The SPE scheme tailored for our proposed schemes are described in Alg. 1, where d stands for the maximal number of deletions between two consecutive search queries.

- $\mathsf{GenKey}(\mathsf{lks}^{(w)}, \tau) \rightarrow (\mathsf{key}^{(\tau)})$: It takes the local key share $\mathsf{lks}^{(w)}$ of keyword w and the tag τ of keyword/file pair (w, f) as input, and generates the secret key $\mathsf{key}^{(\tau)}$.
- $\mathsf{IncPun}(\mathsf{lks}^{(w)}, \tau) \rightarrow (\mathsf{lks}'^{(w)}, \mathsf{pks}^{(\tau)})$: To delete a keyword/file pair (w, f) associated with tag τ, it punctures current local key share $\mathsf{lks}^{(w)}$ on the tag, and generates the public key share $\mathsf{pks}^{(\tau)}$ and the new local key share $\mathsf{lks}'^{(w)}$.

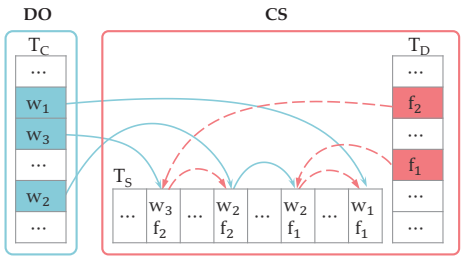


Fig. 3. The Hybrid index in FLY. The green solid line denotes a search list and the red dashed line denotes a deletion list.

- $\mathsf{RegenKey}(\mathsf{SK}, \tau) \rightarrow \{\mathsf{key}^{(\tau)}, \perp\}$: It takes key shares SK and the tag τ as input, and regenerates the secret key $\mathsf{key}^{(\tau)}$ if and only if tag τ has never been punctured before.

SPE is proven to be selectively secure (IND-sPUN-CPA secure), and its security can be reduced to the irretrievability of the secret keys relevant to deleted pairs. We provide an example to illustrate the high-level idea of SPE in Appendix A, and refer the readers to [28] and [30] for complete explanations and security proofs of SPE and PPRF, respectively.

3 An Efficient FP DSSE Scheme

3.1 Building Blocks

To speed up the search process, there are two kinds of approaches to design a searchable index: the forward index that establishes keyword lists per file, and the inverted index that builds file identifer lists per keyword. The forward index more easily supports updates, but requires the search time to grow linearly with the number of files. By contrast, the inverted index achieves sublinear search time, but is hard to support updates, especially deletion. To support efficient searches and updates, we propose a flexible index, Hybrid, which incorporates the merits of inverted and forward indexes, while compacting the index size.

Hybrid is composed of a set of *hybrid lists*. For each keyword $w \in \mathsf{W}$, a hybrid list HL_w links a set of hybrid nodes, where each node not only contains the information about a search list L_w, but also contains the information about a deletion list L_f for $f \in \mathsf{DB}(w)$. Formally, the hybrid list can be defined in a hierarchical way as follows:

Definition 1 (Search list)**.** *For each keyword $w \in \mathsf{W}$, a search list L_w links a set of search nodes and each node is defined as $\mathsf{N} = (f, pos(\mathsf{N}^+), key(\mathsf{N}^+))$, where $f \in \mathsf{DB}(w)$ is a file identifier, $pos(\mathsf{N}^+)$ is the position of the successor node in L_w, and $key(\mathsf{N}^+)$ is the secret key to decrypt the successor node. In the special case, if N is the end node of L_w, we have $pos(\mathsf{N}^+) = \mathbf{0}$.*

Definition 2 (Deletion list)**.** *For each file identifier $f \in \mathsf{DB}$, a deletion list L_f links a set of deletion nodes and each node is defined as $\widetilde{\mathsf{N}} = (w, pos(\widetilde{\mathsf{N}}^+))$, where $w \in \mathsf{DB}(f)$ is a keyword, and $pos(\widetilde{\mathsf{N}}^+)$ is the position of the successor node in L_f. In the special case, if $\widetilde{\mathsf{N}}$ is the end node of L_f, we have $pos(\widetilde{\mathsf{N}}^+) = \mathbf{0}$.*

Definition 3 (Hybrid list)**.** *For each keyword $w \in \mathsf{W}$, a hybrid list HL_w links a set of hybrid nodes and each node is defined as $\mathsf{HN} = (f, pos(\mathsf{HN}^+), key(\mathsf{HN}^+), pos(\widetilde{\mathsf{N}}^+))$, where $f \in \mathsf{DB}(w)$ is a file identifier, $pos(\mathsf{HN}^+)$ is the position of the successor node in HL_w, $key(\mathsf{HN}^+)$ is the secret key to decrypt the successor node, and $pos(\widetilde{\mathsf{N}}^+)$ is the position of the successor in the deletion list L_f. For node HN, its first three elements are from node $\mathsf{N} =$*

Protocol 1 FLY

FLY.Setup

$\mathbf{DO}(\lambda) \rightarrow (\sigma, \mathsf{EDB})$

$(k_{\mathsf{pos}}, k_1, k_2, k_3) \xleftarrow{\$} \{0,1\}^\lambda$ $(\mathrm{T_C}, \mathrm{T_D}, \mathrm{T_S}) \leftarrow$ empty map
$\sigma \leftarrow (k_{\mathsf{pos}}, k_1, k_2, k_3, \mathrm{T_C})$; $\mathsf{EDB} \leftarrow (\mathrm{T_D}, \mathrm{T_S})$

FLY.Addition

$\mathbf{DO}(\sigma, f, \mathsf{DB}(f)) \rightarrow (\sigma', \tau_{\mathsf{add}})$

$\mathsf{AST} \leftarrow \emptyset$; $c \leftarrow 1$
for each keyword $w \in \mathsf{DB}(f)$ **do**
  if $\mathrm{T_C}[w] = \perp$ **then**
    $\mathsf{head}^{(w)} \leftarrow \mathbf{0}$; $\mathsf{key}^{(w)} \leftarrow \mathbf{0}$
    $\{f$ is the first file containing keyword $w\}$
  else
    $(\mathsf{head}^{(w)}, \mathsf{key}^{(w)}) \leftarrow \mathrm{T_C}[w]$
  if $c = 1$ **then**
    $\mathsf{head}^{(f)} \leftarrow \mathbf{0}$
    $\{w$ is the first keyword of file $f\}$
  $\mathsf{nhead}^{(w)} \leftarrow Z_{k_{\mathsf{pos}}}(w \| f)$; $\mathsf{nkey}^{(w)} \xleftarrow{\$} \{0,1\}^\lambda$
  $\mathsf{HN} \leftarrow (f, \mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{head}^{(f)})$; $r \xleftarrow{\$} \{0,1\}^\lambda$
  $e_1 \leftarrow \mathsf{HN}[1] \oplus H_1(\mathsf{nkey}^{(w)}, r)$
  $e_2 \leftarrow \mathsf{HN}[2] \oplus H_2(\mathsf{nkey}^{(w)}, r)$
  $e_3 \leftarrow \mathsf{HN}[3] \oplus H_3(\mathsf{nkey}^{(w)}, r)$
  $e_4 \leftarrow \mathsf{HN}[4] \oplus H_4(P_{k_3}(f), r)$; $c \leftarrow c + 1$
  $\mathsf{E} \leftarrow (e_1, e_2, e_3, e_4)$; $\mathsf{AST} \leftarrow \mathsf{AST} \cup (\mathsf{nhead}^{(w)}, \mathsf{E}, r)$
  $\mathsf{head}^{(f)} \leftarrow \mathsf{nhead}^{(w)}$; $\mathrm{T_C}[w] \leftarrow (\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$
$\tau_{\mathsf{add}} \leftarrow (\mathsf{AST}, F_{k_1}(f), G_{k_2}(f) \oplus \mathsf{head}^{(f)})$

$\mathbf{CS}(\mathsf{EDB}, \tau_{\mathsf{add}}) \rightarrow (\mathsf{EDB}')$

parse τ_{add} as $(\mathsf{AST}, \tau_1, \tau_2)$
$\mathrm{T_D}[\tau_1] \leftarrow \tau_2$
for each tuple $(\mathsf{nhead}^{(w)}, \mathsf{E}, r) \in \mathsf{AST}$ **do**
  $\mathrm{T_S}[\mathsf{nhead}^{(w)}] \leftarrow (\mathsf{E}, r)$

FLY.Deletion

$\mathbf{DO}(\sigma, f) \rightarrow (\tau_{\mathsf{del}})$

$\tau_{\mathsf{del}} \leftarrow (F_{k_1}(f), G_{k_2}(f), P_{k_3}(f))$

$\mathbf{CS}(\tau_{\mathsf{del}}, \mathsf{EDB}) \rightarrow (\mathsf{EDB}')$

parse τ_{del} as (τ_1, τ_2, τ_3)
if $\mathrm{T_D}[\tau_1] \neq \perp$ **then**
  $\mathsf{curr}^{(f)} \leftarrow \mathrm{T_D}[\tau_1] \oplus \tau_2$; $\mathrm{T_D}[\tau_1] \leftarrow \perp$
  while $\mathsf{curr}^{(f)} \neq \mathbf{0}$ **do**
    $(\mathsf{E}, r) \leftarrow \mathrm{T_S}[\mathsf{curr}^{(f)}]$; $\mathsf{E}[1] \leftarrow \perp$
    $\mathrm{T_S}[\mathsf{curr}^{(f)}] \leftarrow (\mathsf{E}, r)$; $\mathsf{curr}^{(f)} \leftarrow \mathsf{E}[4] \oplus H_4(\tau_3, r)$

FLY.Search

$\mathbf{DO}(\sigma, w) \rightarrow (\Omega)$

$\Omega \leftarrow \mathrm{T_C}[w]$

$\mathbf{AC}(\Omega, w) \rightarrow (\tau_{\mathsf{srh}})$

$\tau_{\mathsf{srh}} \leftarrow \Omega$

$\mathbf{CS}(\tau_{\mathsf{srh}}, \mathsf{EDB}) \rightarrow (\mathsf{IND})$

parse τ_{srh} as $(\mathsf{head}^{(w)}, \mathsf{key}^{(w)})$; $\mathsf{IND} \leftarrow$ empty set
$\mathsf{newHead}^{(w)} \leftarrow \perp$; $\mathsf{newKey}^{(w)} \leftarrow \perp$; $\mathsf{prevPos}^{(w)} \leftarrow \mathbf{0}$
$\mathsf{curPos}^{(w)} \leftarrow \mathsf{head}^{(w)}$; $\mathsf{curKey}^{(w)} \leftarrow \mathsf{key}^{(w)}$
while $\mathsf{curPos}^{(w)} \neq \mathbf{0}$ **do**
  $(\mathsf{E}, r) \leftarrow \mathrm{T_S}[\mathsf{curPos}^{(w)}]$
  $\mathsf{nextPos}^{(w)} \leftarrow \mathsf{E}[2] \oplus H_2(\mathsf{curKey}^{(w)}, r)$
  $\mathsf{nextKey}^{(w)} \leftarrow \mathsf{E}[3] \oplus H_3(\mathsf{curKey}^{(w)}, r)$
  if $\mathsf{E}[1] = \perp$ **then**
    ListRemove($\mathrm{T_S}, \mathsf{nextPos}^{(w)}, \mathsf{nextKey}^{(w)}, \mathsf{curPos}^{(w)}, \mathsf{curKey}^{(w)}, \mathsf{prevPos}^{(w)}$)
  else
    $f \leftarrow \mathsf{E}[1] \oplus H_1(\mathsf{curKey}^{(w)}, r)$; $\mathsf{IND} \leftarrow \mathsf{IND} \cup \{f\}$
    $\mathsf{prevPos}^{(w)} \leftarrow \mathsf{curPos}^{(w)}$
  $\mathsf{curPos}^{(w)} \leftarrow \mathsf{nextPos}^{(w)}$; $\mathsf{curKey}^{(w)} \leftarrow \mathsf{nextKey}^{(w)}$

Algorithm 2 ListRemove($\mathrm{List}, \mathsf{nPos}, \mathsf{nKey}, \mathsf{cPos}, \mathsf{cKey}, \mathsf{pPos}$)

$\mathrm{List}[\mathsf{cPos}] \leftarrow \perp$; $(\mathsf{E}^-, r^-) \leftarrow \mathrm{List}[\mathsf{pPos}]$
$\mathsf{E}^-[2] \leftarrow \mathsf{E}^-[2] \oplus \mathsf{cPos} \oplus \mathsf{nPos}$
$\mathsf{E}^-[3] \leftarrow \mathsf{E}^-[3] \oplus \mathsf{cKey} \oplus \mathsf{nKey}$
$\mathrm{List}[\mathsf{pPos}] \leftarrow (\mathsf{E}^-, r^-)$

$(f, pos(\mathsf{N}^+), key(\mathsf{N}^+))$ *of the search list* L_w*, and its last element is from the node* $\widetilde{\mathsf{N}} = (w, pos(\widetilde{\mathsf{N}}^+))$ *of the deletion list* L_f.

In the hybrid list HL_w, the hybrid node regarding the keyword/file pair (w, f) keeps the location/key pair for the successor node in the search list L_w, and the location of the next node in the deletion list L_f. As for storage structure, the CS uses a search map $\mathrm{T_S}$ to store all hybrid nodes while keeping the locations for the head nodes of deletion lists in a deletion map $\mathrm{T_D}$, and the DO keeps the location/key pairs for the head nodes of hybrid lists in a local map $\mathrm{T_C}$.

Consider a database that consists of four keyword/file pairs $\{(w_1, f_1), (w_2, f_1), (w_2, f_2), (w_3, f_2)\}$. A sample Hybrid index is shown in Fig. 3, where search lists make for sublinear search time, and deletion lists allow for flexible updates. For example, to search keyword w_2, a search token τ_{srh} consisting of the location/key pair of the head of the hybrid list HL_{w_2} is sent to the CS, which first locates the head of the search list L_{w_2} and then finds all the files containing keyword w by successively visiting L_{w_2}. Similarly, to delete file f_1, a deletion token τ_{del} consisted of the location of the head of the deletion list L_{f_1} is sent to the CS, which visits L_{f_1} in order to find all the keywords belonging to file f_1.

3.2 Construction of FLY

Let $Z : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$, $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$, $G : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ and $P : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be PRFs. Let $H_i : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^\lambda (i = 1, 2, 3, 4)$ be keyed hash functions. Based on the index Hybrid, FLY is constructed as shown in Protocol 1.

• FLY.Setup. The DO maintains a state σ consisted of PRF keys and a local map $\mathrm{T_C}$ that stores the location/key pairs for head nodes of hybrid lists. Besides, the DO creates the encrypted database EDB consisted of a deletion map $\mathrm{T_D}$ and a search map $\mathrm{T_S}$, and sends EDB to the CS.

• FLY.Addition. To add a file f into EDB, the DO performs as follows: For each keyword $w \in \mathsf{DB}(f)$, it creates a node $\mathsf{HN} = (f, \mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{head}^{(f)})$ as the new head of HL_w according to Definition 3. It then encrypts the first three elements of HN related to the search list L_w with a fresh key $\mathsf{nkey}^{(w)}$ for forward privacy, while encrypts the fourth element of HN related to the deletion list L_f with a keyed PRF. Furthermore, it calculates a new location $\mathsf{nhead}^{(w)}$ for node HN and updates $\mathrm{T_C}[w]$ with $(\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$.

On receiving the addition token τ_{add}, the CS updates EDB as follows: It updates the deletion map $\mathrm{T_D}$ to point to the new head of the deletion list L_f. For each keyword $w \in \mathsf{DB}(f)$, it stores the new head of the hybrid list HL_w into location $\mathsf{nhead}^{(w)}$ of the search map $\mathrm{T_S}$.

• FLY.Deletion. To delete a file f, the DO sends a token τ_{del} to the CS, which sets the first elements of nodes related to the deletion list L_f to NULL. FLY adopts a *lazy deletion* mechanism that delays node releasing to the search phase.

• FLY.Search. To search a keyword w, the AC first requests a search certificate Ω from the DO, where Ω contains the location/key pair of current head of the hybrid list HL_w[1,2]. Once authorized, the AC sends Ω as the search token τ_{srh} to the CS, which locates nodes of HL_w in turn until reaching the end. If a node's first element is non-empty, the CS recovers the corresponding file identifier and puts it into the search result IND. Otherwise, the CS removes this node from HL_w and releases its space by running the subroutine in Alg. 2. It is worth noticing that FLY enables the CS to store the encrypted hybrid lists after the search phase without requiring the DO to re-encrypt the search results.

3.3 Discussion

FLY allows the AC to keep a small-sized search certificate, and generate search tokens in a lightweight way. Thus, we mainly discuss how FLY realizes the other design goals. In addition, we consider how to improve FLY so as to reduce the authorization interactions between the DO and AC.

Authorized search. FLY encrypts each search list separately, disenabling the AC to search undelegated keywords. Therefore, we focus on the case that the AC searches the data outsourced after authorization. Suppose that the DO grants the search certificate $\Omega = \mathrm{T_C}[w]$ to the AC at time t_1, and later the DO adds a new pair (w, f) into the database at time t_2 ($t_2 > t_1$). According to algorithm FLY.Addition, the DO constructs a node HN as the new head of the hybrid list HL_w, chooses a new location/key pair $(\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$ for HN, and updates $\mathrm{T_C}[w]$ with $(\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$. Since the update happens after authorization, the AC cannot obtain the location/key pair $(\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$ of the new head, and cannot generate new search tokens. If the "old" search tokens generated by Ω can search the newly added data, then forward privacy is violated. In other word, authorized search is guaranteed by forward privacy.

Flexibility. Although Protocol 1 only shows the process of file-based (coarse-grained) updates, FLY can be easily extended to support pair-based (fine-grained) updates: To delete a pair (w, f), the DO sends a token $\tau_{\mathsf{del}} = Z_{k_{\mathsf{pos}}}(w \| f)$ to the CS, which sets the first element of the node located at $\mathrm{T_S}[\tau_{\mathsf{del}}]$ to NULL. To add a pair (w, f), the DO sends a token $\tau_{\mathsf{add}} = (\mathsf{nhead}^{(w)}, \mathsf{E}, r, F_{k_1}(f), G_{k_2}(f) \oplus \mathsf{nhead}^{(w)})$ to the CS, where the encrypted hybrid node E is defined similarly as FLY.Addition except that $\mathsf{E}[4] = G_{k_2}(f) \oplus H_4(P_{k_3}(f), r)$. Given the addition token $\tau_{\mathsf{add}} = (\mathsf{nhead}^{(w)}, \mathsf{E}, r, \tau_1, \tau_2)$, the CS updates the search list by setting $\mathrm{T_S}[\mathsf{nhead}^{(w)}] \leftarrow (\mathsf{E}, r)$ after updating $\mathsf{E}[4]$ as $\mathsf{E}[4] \oplus \mathrm{T_D}[\tau_1]$, and updates the deletion list by $\mathrm{T_D}[\tau_1] \leftarrow \tau_2$. Note that, if the DO wants to modify a keyword w contained in file f into w', it will first delete the pair (w, f) from hybrid list HL_w and then add the pair (w', f) as the new head of hybrid list $\mathrm{HL}_{w'}$.

1. The DO may add a dummy head for the keyword to avoid the situation that the real head is deleted.

2. The AC can request the search certificate for a set of keywords at once. For ease of illustration, we only consider the single keyword case.

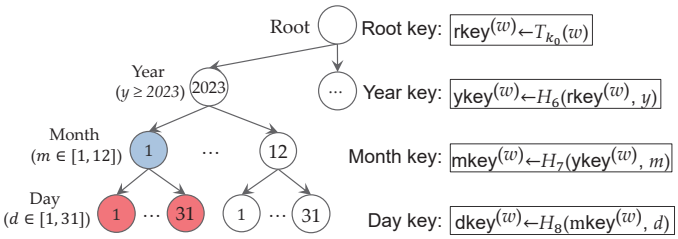


Fig. 4. A time-based key tree of keyword w. From up to down, a node at each layer has a root key, year key, month key, day key, respectively.

Reducing authorization interactions. In FLY, the AC can only query the data outsourced before authorization, and needs to contact the DO for a new search certificate if it wants to access the newly added data. In our application scenario, the DO periodically uploads the data incrementally collected by IoT devices. Therefore, FLY makes the DO stay online distributing search certificates and become a performance bottleneck. A preferable solution is to grant each AC an eligible time period, within which the AC can search designated keywords without interacting with the DO. Inspired by our previous work [31], we achieve time-based authorization by building a time-based key tree $\mathcal{T}_w$ for each keyword w, where each tree node $\mathcal{N}$ is associated with a key, denoted by $\mathcal{N}.key$. In practice, the time granularity can be adjusted according to the data generation rate. For simplicity, we assume that the time granularity is a day, and consider a four-layer time-based key tree as shown in Fig. 4.

Let $T : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be a PRF, let $H_i : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^\lambda (i = 6, 7, 8, 9)$ be keyed hash functions, and let (y, m, d), (y, m), and (y) denote a particular day, month, and year, respectively. Based on the time-based key trees, FLY can be extended as follows:

• Setup. This algorithm is similar to FLY.Setup, except that the local state σ contains a PRF key $k_0 \in \{0,1\}^\lambda$ in addition, and for each keyword w, the local map $\mathrm{T_C}$ extra stores a keyword counter $c^{(w)}$, which denotes the number of files containing this keyword in current date.

• Addition. To add a file f in date (y, m, d), the DO performs in a similar way as FLY.Addition, except that for each keyword $w \in \mathsf{DB}(f)$, the fresh key $\mathsf{nkey}^{(w)}$ and the new location $\mathsf{nhead}^{(w)}$ are calculated by Eq. (1):

$$\begin{aligned} \mathsf{nkey}^{(w)} &\leftarrow H_9(\mathsf{dkey}^{(w)}, c^{(w)} || 0) \\ \mathsf{nhead}^{(w)} &\leftarrow H_9(\mathsf{dkey}^{(w)}, c^{(w)} || 1) \end{aligned} \tag{1}$$

where $\mathsf{dkey}^{(w)}$ as current day key of keyword w is calculated in a hierarchically way as shown in Fig. 4. Besides, the DO updates $\mathrm{T_C}[w]$ with $(\mathsf{nkey}^{(w)}, \mathsf{nhead}^{(w)}, c^{(w)} + 1)$.

As time is accurate to a day, we assume that the DO uploads newly collected data at the beginning of each day, and that all the keywords will be updated in a day. Once all the files have been uploaded, the DO also uploads the encrypted keyword counters and clears them from the local map after outsourcing. Specifically, for each keyword $w \in \mathsf{W}$, the DO chooses a random string $r^{(w)} \in \{0,1\}^\lambda$, encrypts the counter by calculating

$e^{(w)} \leftarrow c^{(w)} \oplus H_9(\mathsf{dkey}^{(w)}, r^{(w)})$, and generates a label by calculating $l^{(w)} \leftarrow H_9(\mathsf{dkey}^{(w)}, y||m||d)$. The additional outsourced data is in the form of $\{(l^{(w)}, e^{(w)}, r^{(w)})\}_{w \in \mathsf{W}}$.

• Deletion. This algorithm is the same as FLY.Deletion.

• Authorization. If the AC is authorized to search keyword w during time period $[s, t]$, the DO first finds the maximal covering nodes MCN in the time-based key tree $\mathcal{T}_w$, and then sets the search certificate as the keys associated with all these nodes, i.e., $\Omega \leftarrow \{\mathcal{N}.key\}_{\mathcal{N} \in \mathsf{MCN}}$. For example, if the eligible time period is $[(2023, 1, 1), (2023, 1, 31)]$, the maximal covering node is marked by blue in Fig. 4, and the search certificate is actually a month key, i.e., $\Omega = \{\mathsf{mkey}^{(w)}\}$, where $\mathsf{mkey}^{(w)} = H_7(\mathsf{ykey}^{(w)}, 1)$, $\mathsf{ykey}^{(w)} = H_6(\mathsf{rkey}^{(w)}, 2013)$, and $\mathsf{rkey}^{(w)} = T_{k_0}(w)$.

• Search. To search a keyword w on date (y, m, d), the AC first uses its search certificate Ω to generate the corresponding day key $\mathsf{dkey}^{(w)}$, and sets the search token as $\tau_{srh} = (\mathsf{dkey}^{(w)})$. Given the search token, the CS first locates the counter ciphertext through the label $l^{(w)}$ and then recovers the counter plaintext by calculating $c^{(w)} \leftarrow e^{(w)} \oplus H_9(\mathsf{dkey}^{(w)}, r^{(w)})$. Next, the CS calculates the location and key of the current header of hybrid list HL_w by using Eq. (1), and then performs in the same way as FLY.Search to update hybrid list HL_w and generate the search result IND.

Note that, if the eligible time period $[s, t]$ covers date (y, m, d), the AC possesses either the day key or a key associated with an ancestor node in tree $\mathcal{T}_w$, and thus can generate search token correctly. For example, the AC with a month key $\mathsf{mkey}^{(w)}$ for $(2023, 1)$ can search keyword w within $[(2023, 1, 1), (2023, 1, 31)]$ by producing day keys $H_8(\mathsf{mkey}^{(w)}, 1), \ldots, H_8(\mathsf{mkey}^{(w)}, 31)$. In terms of authorized search, the AC cannot infer the keys associated with nodes outside the eligible period due to the security of hash functions and PRFs. As for flexibility, current extension supports only file-based updates, but can be improved by associating each hybrid node with label $\mathsf{label}^{(w)} = Z_{k_{\mathsf{pos}}}(w \| f)$.

4 An Efficient FP and BP DSSE Scheme

FLY fails to support secure deletion, since a search token can decrypt all files containing a specified keyword that are uploaded before authorization. In other word, if the CS backups the deleted files, then it still can recover them in the search phase. To address this, we employ SPE for backward privacy, rendering the deleted files that haven't been searched yet to be inaccessible. However, existing SPE-based schemes cache search results and support only fine-grained updates. To solve this problem, we propose a dual-key mechanism consisted of a primary key and a secondary key, which can be punctured by SPE or PPRFs, respectively. For ease of illustration, we first provide a basic construction, FLY+, which utilizes SPE to puncture the primary key achieving all our design goals except for flexibility. Base on this, we present the full-featured version, FLY++, which utilizes SPE and PPRFs to puncture keys on demand.

4.1 FLY+: The Basic Construction

Let $\mathsf{SPE} = (\mathsf{GenKey}, \mathsf{IncPun}, \mathsf{RegenKey})$ be a SPE scheme as defined in Section 2.4. Let $Z : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$, $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$, $G : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$, and $P : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\mu$ be PRFs, where μ is related to the size of tag space. Let $H_i : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^\lambda (i = 1, 2, 3)$ and $H_4 : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^\kappa$ be keyed hash functions, where κ is related to the length of tags μ and the security parameter λ. The details of FLY+ are shown in Protocol 2, where the deletion lists related to file-based updates is excluded.

• FLY+.Setup. Compared to FLY, the DO additionally keeps PRF keys $k_{\mathsf{pun}}, k_{\mathsf{tag}}, k_{\mathsf{spe}}$ related to SPE, and maintains the authorization counter $\mathsf{ac}^{(w)}$ and the local key share $\mathsf{lks}^{(w)}$ for each keyword w in the local map $\mathrm{T_C}$. Besides, EDB includes a puncture map $\mathrm{T_P}$ to store the encrypted public key shares relevant to SPE in addition.

• FLY+.Addition. To add a pair (w, f), the DO calculates a tag τ, and constructs a node $\mathsf{HN} = (f, \mathsf{head}^{(w)}, \mathsf{key}^{(w)})$ as the new head of HL_w. For backward privacy, the DO encrypts f with the dual-key mechanism, where the primary key $\mathsf{key}^{(\tau)}$ is generated by the SPE.GenKey algorithm, and the secondary key $\mathsf{nkey}^{(w)}$ is randomly chosen. For forward privacy, the DO encrypts the remaining elements with the secondary key $\mathsf{nkey}^{(w)}$ and the tag τ. In terms of updating the local map, the DO calculates a new location $\mathsf{nhead}^{(w)}$ for node HN and updates the location/key pair in $\mathrm{T_C}[w]$ with $(\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$, while keeping the authorization counter $\mathsf{ac}^{(w)}$ and the local key share $\mathsf{lks}^{(w)}$ unchanged. Note that, if f is the first file of keyword w, $\mathsf{ac}^{(w)}$ and $\mathsf{lks}^{(w)}$ need to be initialled with 0 and $G_{k_{\mathsf{spe}}}(w||0)$, respectively. Given the addition token τ_{add}, the CS stores the new head of HL_w into location $\mathsf{nhead}^{(w)}$ of the search map $\mathrm{T_S}$.

• FLY+.Deletion. To delete a pair (w, f), the DO punctures the local key share $\mathsf{lks}^{(w)}$ on tag τ by using algorithm SPE.IncPun to produce a new local key share $\mathsf{lks}'^{(w)}$ and a public key share $\mathsf{pks}^{(\tau)}$. It updates the local map $\mathrm{T_C}[w]$ with $\mathsf{lks}'^{(w)}$, and then sends a deletion token $\tau_{\mathsf{del}} = (\mathsf{pos}, e)$ to the CS, where pos is the position of the deleted pair in the search map, and e is the encrypted public key share by key $\mathsf{pun}^{(w)}$. Given τ_{del}, the CS sets the first element of the node in $\mathrm{T_S}[\mathsf{pos}]$ to NULL, and sets $\mathrm{T_P}[\mathsf{pos}]$ to e.

• FLY+.Search. Compared to FLY.Search, the search certificate Ω additionally contains $(\mathsf{lks}^{(w)}, \mathsf{pun}^{(w)})$, the keys related to SPE. After authorization, the DO needs to update the local map $\mathrm{T_C}$ accordingly, where the authorization counter $\mathsf{ac}^{(w)}$ is increased by 1 and the local key share is calculated by $\mathsf{lks}^{(w)} \leftarrow G_{\mathsf{spe}}(w||\mathsf{ac}^{(w)})$. That is, both $\mathsf{ac}^{(w)}$ and $\mathsf{lks}^{(w)}$ will be changed after each search authorization. To search a keyword w, the AC sends Ω as the search token τ_{srh} to the CS, which locates nodes of the hybrid list HL_w in turn until reaching the end. The main difference from FLY is that, for each hybrid node, if the first element is NULL, the CS recovers the corresponding public key share with key $\mathsf{pun}^{(w)}$ and adds them into a set PKS, otherwise it adds the encrypted file identifier together with auxiliary information into a set ES. Once completing traverse, the CS combines the local key share $\mathsf{lks}^{(w)}$ with PKS to produce set SK, which can be used to regenerate primary keys by algorithm SPE.RegenKey for decrypting those undeleted pairs in ES.

Note that, for an available node, its primary key is calculated only once (when the node is accessed for the first time) and will be kept in the hybrid list for future use, while its secondary key is revealed in the search phase. Therefore, no re-encryption and caching operations are required.

Protocol 2 FLY+

FLY+.Setup

$\mathbf{DO}(\lambda) \to (\sigma, \mathsf{EDB})$

$(k_{\mathsf{pos}}, k_{\mathsf{pun}}, k_{\mathsf{tag}}, k_{\mathsf{spe}}) \xleftarrow{\$} \{0,1\}^{\lambda}$
$(\mathrm{T_C}, \mathrm{T_S}, \mathrm{T_P}) \leftarrow$ empty map; $\mathsf{EDB} \leftarrow (\mathrm{T_S}, \mathrm{T_P})$
$\sigma \leftarrow (k_{\mathsf{pos}}, k_{\mathsf{pun}}, k_{\mathsf{tag}}, k_{\mathsf{spe}}, \mathrm{T_C})$

FLY+.Addition

$\mathbf{DO}(\sigma, (w, f)) \to (\sigma', \tau_{\mathsf{add}})$

if $\mathrm{T_C}[w] = \perp$ **then**
  $(\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}^{(w)}) \leftarrow (\mathbf{0}, \mathbf{0}, 0, G_{k_{\mathsf{spe}}}(w||0))$
else
  $(\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}^{(w)}) \leftarrow \mathrm{T_C}[w]$
$\mathsf{nhead}^{(w)} \leftarrow Z_{k_{\mathsf{pos}}}(w\|f)$; $\mathsf{nkey}^{(w)} \xleftarrow{\$} \{0,1\}^{\lambda}$
$\tau \leftarrow P_{k_{\mathsf{tag}}}(w\|f)$; $\mathsf{pkey} \leftarrow \mathbf{0}$
$\mathsf{key}^{(\tau)} \leftarrow \mathsf{SPE.GenKey}(\mathsf{lks}^{(w)}, \tau)$
$\mathsf{HN} \leftarrow (f, \mathsf{head}^{(w)}, \mathsf{key}^{(w)})$
$e_1 \leftarrow \mathsf{HN}[1] \oplus H_1(\mathsf{nkey}^{(w)}, \mathsf{key}^{(\tau)})$
$e_2 \leftarrow \mathsf{HN}[2] \oplus H_2(\mathsf{nkey}^{(w)}, \tau)$
$e_3 \leftarrow \mathsf{HN}[3] \oplus H_3(\mathsf{nkey}^{(w)}, \tau)$
$\mathsf{E} \leftarrow (e_1, e_2, e_3)$
$\mathrm{T_C}[w] \leftarrow (\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}^{(w)})$
$\tau_{\mathsf{add}} \leftarrow (\mathsf{nhead}^{(w)}, \mathsf{E}, \tau, \mathsf{pkey})$

$\mathbf{CS}(\mathsf{EDB}, \tau_{\mathsf{add}}) \to (\mathsf{EDB}')$

parse τ_{add} as $(\mathsf{nhead}^{(w)}, \mathsf{E}, \tau, \mathsf{pkey})$
$\mathrm{T_S}[\mathsf{nhead}^{(w)}] \leftarrow (\mathsf{E}, \tau, \mathsf{pkey})$

FLY+.Deletion

$\mathbf{DO}(\sigma, (w, f)) \to (\tau_{\mathsf{del}})$

$(\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}^{(w)}) \leftarrow \mathrm{T_C}[w]$
$\tau \leftarrow P_{k_{\mathsf{tag}}}(w\|f)$; $(\mathsf{lks}'^{(w)}, \mathsf{pks}^{(\tau)}) \leftarrow \mathsf{SPE.IncPun}(\mathsf{lks}^{(w)}, \tau)$
$\mathsf{pun}^{(w)} \leftarrow F_{k_{\mathsf{pun}}}(w\|\mathsf{ac}^{(w)})$; $e \leftarrow \mathsf{pks}^{(\tau)} \oplus H_4(\mathsf{pun}^{(w)}, \tau)$
$\mathrm{T_C}[w] \leftarrow (\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}'^{(w)})$
$\mathsf{pos} \leftarrow Z_{k_{\mathsf{pos}}}(w\|f)$; $\tau_{\mathsf{del}} \leftarrow (\mathsf{pos}, e)$

$\mathbf{CS}(\mathsf{EDB}, \tau_{\mathsf{del}}) \to (\mathsf{EDB}')$

parse τ_{del} as (pos, e); $(\mathsf{E}, \tau) \leftarrow \mathrm{T_S}[\mathsf{pos}]$
$\mathsf{E}[1] \leftarrow \perp$; $\mathrm{T_S}[\mathsf{pos}] \leftarrow (\mathsf{E}, \tau)$; $\mathrm{T_P}[\mathsf{pos}] \leftarrow e$

FLY+.Search

$\mathbf{DO}(\sigma, w) \to (\sigma', \Omega)$

$(\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}^{(w)}) \leftarrow \mathrm{T_C}[w]$
$\mathsf{pun}^{(w)} \leftarrow F_{k_{\mathsf{pun}}}(w\|\mathsf{ac}^{(w)})$; $\mathsf{ac}^{(w)} \leftarrow \mathsf{ac}^{(w)} + 1$
$\Omega \leftarrow (\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{lks}^{(w)}, \mathsf{pun}^{(w)})$
$\mathrm{T_C}[w] \leftarrow (\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{ac}^{(w)}, G_{k_{\mathsf{spe}}}(w||\mathsf{ac}^{(w)}))$

$\mathbf{AC}(\Omega, w) \to (\tau_{\mathsf{srh}})$

$\tau_{\mathsf{srh}} \leftarrow \Omega$

$\mathbf{CS}(\tau_{\mathsf{srh}}, \mathsf{EDB}) \to (\mathsf{IND})$

parse τ_{srh} as $(\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{lks}^{(w)}, \mathsf{pun}^{(w)})$
$(\mathrm{PKS}, \mathrm{ES}, \mathsf{IND}) \leftarrow$ empty set
$\mathsf{newHead}^{(w)} \leftarrow \perp$; $\mathsf{newKey}^{(w)} \leftarrow \perp$; $\mathsf{prevPos}^{(w)} \leftarrow \mathbf{0}$
$\mathsf{curPos}^{(w)} \leftarrow \mathsf{head}^{(w)}$; $\mathsf{curKey}^{(w)} \leftarrow \mathsf{key}^{(w)}$
while $\mathsf{curPos}^{(w)} \neq \mathbf{0}$ **do**
  $(\mathsf{E}, \tau, \mathsf{pkey}) \leftarrow \mathrm{T_S}[\mathsf{curPos}^{(w)}]$
  $\mathsf{nextPos}^{(w)} \leftarrow \mathsf{E}[2] \oplus H_2(\mathsf{curKey}^{(w)}, \tau)$
  $\mathsf{nextKey}^{(w)} \leftarrow \mathsf{E}[3] \oplus H_3(\mathsf{curKey}^{(w)}, \tau)$
  if $\mathsf{E}[1] = \perp$ **then**
    $e \leftarrow \mathrm{T_P}[\mathsf{curPos}^{(w)}]$; $\mathsf{pks}^{(\tau)} \leftarrow e \oplus H_4(\mathsf{pun}^{(w)}, \tau)$
    $\mathrm{T_P}[\mathsf{curPos}^{(w)}] \leftarrow \perp$; $\mathrm{PKS} \leftarrow \mathrm{PKS} \cup \{\mathsf{pks}^{(\tau)}\}$
    $\mathsf{ListRemove}(\mathrm{T_S}, \mathsf{nextPos}^{(w)}, \mathsf{nextKey}^{(w)}, \mathsf{curPos}^{(w)}, \mathsf{curKey}^{(w)}, \mathsf{prevPos}^{(w)})$
  else
    $\mathrm{ES} \leftarrow \mathrm{ES} \cup (\mathsf{E}, \tau, \mathsf{pkey}, \mathsf{curPos}^{(w)}, \mathsf{curKey}^{(w)})$
    $\mathsf{prevPos}^{(w)} \leftarrow \mathsf{curPos}^{(w)}$
  $\mathsf{curPos}^{(w)} \leftarrow \mathsf{nextPos}^{(w)}$; $\mathsf{curKey}^{(w)} \leftarrow \mathsf{nextKey}^{(w)}$
$\mathsf{SK} \leftarrow (\mathsf{lks}^{(w)}, \mathrm{PKS})$
for each $(\mathsf{E}, \tau, \mathsf{pkey}, \mathsf{curPos}^{(w)}, \mathsf{curKey}^{(w)}) \in \mathrm{ES}$ **do**
  if $\mathsf{pkey} = \mathbf{0}$ **then**
    $\mathsf{key}^{(\tau)} \leftarrow \mathsf{SPE.RegenKey}(\mathsf{SK}, \tau)$
    $\mathsf{pkey} \leftarrow \mathsf{key}^{(\tau)}$; $\mathrm{T_S}[\mathsf{curPos}^{(w)}] \leftarrow (\mathsf{E}, \tau, \mathsf{pkey})$
    {A pair is accessed for the first time}
  $f \leftarrow \mathsf{E}[1] \oplus H_1(\mathsf{curKey}^{(w)}, \mathsf{pkey})$; $\mathsf{IND} \leftarrow \mathsf{IND} \cup \{f\}$

4.2 The Construction of FLY++

Let $\widetilde{F}$ be a PPRF as defined in Section 2.3 with input domain $[n]$, where n is the maximal number of files being outsourced. Let $\mathsf{SE} = (\mathsf{Enc}, \mathsf{Dec})$ be a SE scheme, let $X : \{0,1\}^{\lambda} \times \{0,1\}^{*} \to \{0,1\}^{\lambda}$ be a PRF, and let $H_5 : \{0,1\}^{*} \times \{0,1\}^{*} \to \{0,1\}^{\lambda}$ be a keyed hash function. Given PRFs Z, F, G, P and hash functions $H_1, \ldots, H_4$ as defined in FLY+, the details of FLY++ are shown in Protocol 3. As FLY++ is built based on FLY+, we just describe the differences from FLY+ for briefness.

• FLY++.Setup. The local state σ additionally includes the key sk of PPRF $\widetilde{F}$, the key k_4 of PRF X, and a symmetric key k_5. Besides, the DO locally maintains a deletion map $\mathrm{T_D}$ to record the head node, flag, and file counter for each file, where the flag is converted from 0 into 1 if a file is deleted, and the file counter denotes the order of a file being added.

• FLY++.Addition. To add a pair (w, f), a hybrid node is generated as Definition 3. The main difference from FLY+ is that the file identifier f is encrypted with SE under the dual-key mechanism, where the primary key $\mathsf{key}^{(\tau)}$ is generated in the same way as FLY+, but the secondary key $\mathsf{key}^{(f)}$ is created by a PPRF $\widetilde{F}$. Besides, the DO outsources the encrypted counter of file f, so that the AC can recover the secondary key $\mathsf{key}^{(f)}$ in the search phase. To add a file f, this algorithm is run for each keyword $w \in \mathsf{DB}(f)$.

• FLY++.Deletion. The process of pair-based deletion is the same as FLY+.Deletion. Therefore, Protocol 3 only shows the process of file-based deletion. To delete the file f, the DO sends the head node $\mathsf{head}^{(f)}$ of the deletion list L_f and the key $X_{k_4}(f)$ to the CS, and marks the corresponding entry in $\mathrm{T_D}$ as deleted. The CS performs in a similar way as FLY.Deletion to set the first element of each hybrid node related to the deletion list L_f to NULL.

• FLY++.Search. To search keyword w, the DO employs PPRF to puncture the file set S i.e., the files that have been deleted and those haven't been outsourced, and additionally generates the punctured key k_{S} for the AC. Besides, the DO shares the symmetric key k_5 with the AC at the first authorization. Unlike FLY+.Search letting the CS decrypt the file identifers directly, FLY++.Search makes the CS return

Protocol 3 FLY++

FLY++.Setup

$\mathbf{DO}(\lambda) \rightarrow (\sigma, \mathsf{EDB})$

1: $(k_{\mathsf{pos}}, k_{\mathsf{pun}}, k_{\mathsf{tag}}, k_{\mathsf{spe}}, sk, k_4, k_5) \xleftarrow{\$} \{0,1\}^{\lambda}$
2: $(\mathrm{T_C}, \mathrm{T_S}, \mathrm{T_P}, \mathrm{T_D}) \leftarrow$ empty map; $\mathsf{EDB} \leftarrow (\mathrm{T_S}, \mathrm{T_P})$
3: $\sigma \leftarrow (k_{\mathsf{pos}}, k_{\mathsf{pun}}, k_{\mathsf{tag}}, k_{\mathsf{spe}}, sk, k_4, k_5, \mathrm{T_C}, \mathrm{T_D})$

FLY++.Addition

$\mathbf{DO}(\sigma, (w, f)) \rightarrow (\sigma', \tau_{\mathsf{add}})$

1: Execute lines 1 to 7 of algorithm FLY+.Addition
2: **if** $\mathrm{T_D}[f] = \perp$ **then**
3: $(\mathsf{head}^{(f)}, \mathsf{FLAG}, \mathsf{fc}) \leftarrow (\mathbf{0}, 0, |\mathrm{T_D}| + 1)$
4: **else**
5: $(\mathsf{head}^{(f)}, \mathsf{FLAG}, \mathsf{fc}) \leftarrow \mathrm{T_D}[f]$
6: $\mathsf{HN} \leftarrow (f, \mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{head}^{(f)})$
7: $\mathsf{key}^{(f)} \leftarrow \widetilde{\mathsf{F}}(sk, \mathsf{fc})$; $k_e \leftarrow \mathsf{key}^{(\tau)} \oplus \mathsf{key}^{(f)}$
8: $e_1 \leftarrow \mathsf{SE.Enc}(k_e, \mathsf{HN}[1])$
9: Execute lines 10 to 11 of algorithm FLY+.Addition
10: $e_4 \leftarrow \mathsf{HN}[4] \oplus H_5(X_{k_4}(f), \tau)$; $e_5 \leftarrow \mathsf{SE.Enc}(k_5, \mathsf{fc})$
11: $\mathsf{E} \leftarrow (e_1, e_2, e_3, e_4, e_5)$; $\mathrm{T_D}[f] \leftarrow (\mathsf{nhead}^{(w)}, \mathsf{FLAG}, \mathsf{fc})$
12: Execute lines 13 to 14 of algorithm FLY+.Addition
{The CS performs in the same way as FLY+.Addition}

FLY++.Deletion

$\mathbf{DO}(\sigma, f) \rightarrow (\tau_{\mathsf{del}})$

1: Set FLAG of $\mathrm{T_D}[f]$ to 1; $\tau_{\mathsf{del}} \leftarrow (\mathsf{head}^{(f)}, X_{k_4}(f))$

$\mathbf{CS}(\tau_{\mathsf{del}}, \mathsf{EDB}) \rightarrow (\mathsf{EDB}')$

2: parse τ_{del} as (τ_1, τ_2); $\mathsf{curr}^{(f)} \leftarrow \tau_1$
3: **while** $\mathsf{curr}^{(f)} \neq \mathbf{0}$ **do**
4: $(\mathsf{E}, \tau, \mathsf{pkey}) \leftarrow \mathrm{T_S}[\mathsf{curr}^{(f)}]$; $\mathsf{E}[1] \leftarrow \perp$
5: $\mathrm{T_S}[\mathsf{curr}^{(f)}] \leftarrow (\mathsf{E}, \tau, \mathsf{pkey})$; $\mathsf{curr}^{(f)} \leftarrow \mathsf{E}[4] \oplus H_5(\tau_2, \tau)$

FLY++.Search

$\mathbf{DO}(\sigma, w) \rightarrow (\sigma', \Omega)$

1: Execute lines 1 to 4 of FLY+.Search
2: $\mathsf{S} \leftarrow \{|\mathrm{T_D}| + 1, \ldots, n\}$
3: **for** each non-empty entry $(\mathsf{head}^{(f)}, \mathsf{FLAG}, \mathsf{fc})$ of $\mathrm{T_D}$ **do**
4: **if** $\mathsf{FLAG} = 1$ **then**
5: $\mathsf{S} \leftarrow \mathsf{S} \cup \mathsf{fc}$
6: $k_{\mathsf{S}} \leftarrow \widetilde{\mathsf{F}}.\mathsf{Punc}(sk, \mathsf{S})$; $\Omega \leftarrow \Omega \cup k_{\mathsf{S}}$

$\mathbf{AC}(\Omega, w) \rightarrow (\tau_{\mathsf{srh}})$

7: {The AC performs in the same way as FLY+.Search}

$\mathbf{CS}(\tau_{\mathsf{srh}}, \mathsf{EDB}) \rightarrow (\mathsf{IND})$

8: Perform lines 6 to 22 of algorithm FLY+.Search
9: **for** each $(\mathsf{E}, \tau, \mathsf{pkey}, \mathsf{curPos}^{(w)}, \mathsf{curKey}^{(w)}) \in \mathrm{ES}$ **do**
10: Perform lines 24 to 26 of algorithm FLY+.Search
11: $\mathsf{IND} \leftarrow \mathsf{IND} \cup \{(\mathsf{E}[1], \mathsf{E}[5], \mathsf{pkey})\}$

$\mathbf{AC}(\Omega, k_5, \mathsf{IND}) \rightarrow (\mathsf{IND}')$

12: $\mathsf{IND}' \leftarrow$ empty set
13: **for** each $(\mathsf{E}[1], \mathsf{E}[5], \mathsf{pkey}) \in \mathsf{IND}$ **do**
14: $\mathsf{fc} \leftarrow \mathsf{SE.Dec}(k_5, \mathsf{E}[5])$; $\mathsf{key}^{(f)} \leftarrow \widetilde{\mathsf{F}}.\mathsf{Eval}(k_{\mathsf{S}}, \mathsf{fc})$
15: $k_e \leftarrow \mathsf{pkey} \oplus \mathsf{key}^{(f)}$; $f \leftarrow \mathsf{SE.Dec}(k_e, \mathsf{E}[1])$
16: $\mathsf{IND}' \leftarrow \mathsf{IND}' \cup f$

the encrypted file identifiers, encrypted file counters, and primary keys to the AC, which will recover file identifiers by itself. Specifically, for an undeleted file, the AC can recover the secondary key by running algorithm $\widetilde{\mathsf{F}}.\mathsf{Eval}$ and combines it with the primary key received for decryption.

Time-based authorization. As FLY, FLY++ also requires the DO to persistently distribute new search certificates to ACs. To reduce the DO's workload, the natural way is to make use of the time-based key tree proposed in Section 3.3 to achieve time-based authorization. However, the AC would be unable to recover search results while applying the time-based key tree directly to FLY++. The reason is that compared with FLY, FLY++ generates the search certificate as $\Omega = (\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{pun}^{(w)}, \mathsf{lks}^{(w)}, k_{\mathsf{S}})$, where the extra keys $(\mathsf{pun}^{(w)}, \mathsf{lks}^{(w)}, k_{\mathsf{S}})$ are related to result recovery and will be dynamically updated in the search phase. Therefore, the key challenge lies in how to enable the AC to obtain the latest versions of $(\mathsf{pun}^{(w)}, \mathsf{lks}^{(w)}, k_{\mathsf{S}})$ without interaction with the DO. Our main idea is utilizing the time-based key tree and a shared key to encrypt the latest keys, while outsourcing the ciphertexts, so that the CS can be delegated to distribute keys without compromising privacy. As FLY++ can be extended with the time-based key tree in a similar way as FLY, we focus on answering the following problems: (1) When to generate the latest versions of keys; (2) How to generate these keys; (3) How to encrypt these keys.

As for the first problem, the DO updates all the keys at the beginning of each day, and in the day, it updates the key k_{S} (resp. $\mathsf{lks}^{(w)}$) again once a file (resp. a keyword/file pair) is deleted. Unlike FLY++ that updates the authorization counter of the searching keyword in the search phase, the DO increases the authorization counters of all keywords by 1 only at the beginning of each day. Therefore, the key $\mathsf{pun}^{(w)}$ remains unchanged in a day and needs not to be updated. As for the second problem, the DO generates all the keys in the same way as FLY++. It's worth noting that $\mathsf{lks}^{(w)}$ is generated by calculating PRFs at the beginning of each day, and by running algorithm SPE.IncPun when a keyword/file pair is deleted. In terms of the last problem, the key k_{S} is encrypted with SE under the shared key k_5, and the keys $(\mathsf{pun}^{(w)}, \mathsf{lks}^{(w)})$ are encrypted with SE under the day key corresponding to the date when the key update happens. These ciphertexts will be uploaded to the CS, which will replaces the old ciphertexts with newly received versions. Note that for the key k_{S}, the decryption key k_5 is shared between the DO and the AC, and for the keys $(\mathsf{pun}^{(w)}, \mathsf{lks}^{(w)})$, the AC can obtain the corresponding day key if its eligible time period covers the update date. Therefore, the AC can search designated keywords within the eligible time period without interacting with the DO. Appendix B provides detailed descriptions of this solution.

5 Security Analysis

5.1 $\mathcal{L}_{\prod}$-Adaptive Security of DSSE

Let $\mathcal{A}$ be an adversary, and let $\mathcal{S}$ be a simulator parameterized by leakage functions $\mathcal{L}_{\prod} = (\mathcal{L}_{\prod}^{\mathsf{set}}, \mathcal{L}_{\prod}^{\mathsf{add}}, \mathcal{L}_{\prod}^{\mathsf{del}}, \mathcal{L}_{\prod}^{\mathsf{srh}})$, where each component corresponds to the leakage during Setup, Addition, Deletion, and Search operations in our constructions, respectively. Our DSSE schemes achieve $\mathcal{L}_{\prod}$-adaptive security, with respect to a leakage function $\mathcal{L}_{\prod}$, if the adversary $\mathcal{A}$ can distinguish between **Real** and **Ideal** with only a negligible probability.

• $\mathbf{Real}^{\prod}_{\mathcal{A}}(\lambda)$: Given EDB output by Setup protocol, the adversary $\mathcal{A}$ makes a polynomial number of adaptive queries, and for each query, it receives a search token τ_{srh} by running Search protocol, or an addition token τ_{add} by running Addition protocol, or a deletion token τ_{del} by running Deletion protocol. Finally, $\mathcal{A}$ outputs a bit $b \in \{0,1\}$.

• $\mathbf{Ideal}^{\prod}_{\mathcal{A},\mathcal{S}}(\lambda)$: Given the leakage $\mathcal{L}^{\mathsf{set}}_{\prod}$, the simulator $\mathcal{S}$ generates an encrypted database EDB to the adversary $\mathcal{A}$. $\mathcal{A}$ makes a polynomial number of adaptive queries. Given leakages $\mathcal{L}^{\mathsf{add}}_{\prod}$, $\mathcal{L}^{\mathsf{del}}_{\prod}$, and $\mathcal{L}^{\mathsf{srh}}_{\prod}$, $\mathcal{S}$ returns an appropriate token. Finally, $\mathcal{A}$ outputs a bit $b \in \{0,1\}$.

5.2 Forward and Backward Privacy

The leakage functions $\mathcal{L}_{\prod}$ keep a list Q of all queries issued so far as a state. Each entry of Q is in the form of (v, w) for a search query, or $(v, \mathbf{op}, \mathbf{in})$ for an $\mathbf{op}$ update with input $\mathbf{in}$, where w is a queried keyword, the integer v is a timestamp that is initialized to 0 and incremented by 1 at each query, and $\mathbf{op} \in \{\mathbf{add}, \mathbf{del}\}$ denoting the addition and deletion operations. Given the list Q, for a keyword w, the relevant leakages and definitions can be formally described as:

• $\mathsf{SP}(w) = \{v | (v, w) \in Q\}$, where $\mathsf{SP}(w)$ consists of the timestamps of search queries on keyword w, and thus leaks the repetition of search tokens issued so far;

• $\mathsf{UpHist}(w) = \{(v, \mathbf{op}, f) \in Q\}$, where v is the timestamp of the update, $\mathbf{op} \in \{\mathbf{add}, \mathbf{del}\}$ is the update operation, and f is the updated file identifier, and thus $\mathsf{UpHist}(w)$ consists of the list of all updates on keyword w;

• $\mathsf{TimeDB}(w) = \{(v, f) : (v, \mathbf{add}, (w, f)) \in Q \wedge \forall v', (v', \mathbf{del}, (w, f)) \notin Q\}$, where $\mathsf{TimeDB}(w)$ consists of a list of non-deleted files matching w along with the timestamps of inserting them into the database;

• $\mathsf{DelHist}(w) = \{(v^{\mathbf{add}}, v^{\mathbf{del}}) | \exists f s.t. (v^{\mathbf{add}}, \mathbf{add}, (w, f)) \in Q \wedge (v^{\mathbf{del}}, \mathbf{del}, (w, f)) \in Q\}$ consists of a list of timestamp pairs $(v^{\mathbf{add}}, v^{\mathbf{del}})$, where $v^{\mathbf{add}}$ is the timestamp of adding pair (w, f) and $v^{\mathbf{del}}$ is the timestamp of removing (w, f).

Definition 4 (Forward privacy)**.** *An $\mathcal{L}_{\prod}$-adaptive-secure DSSE scheme achieves forward privacy if there exists a stateless leakage function $\overline{\mathcal{L}}$ s.t $\mathcal{L}^{\mathsf{add}}_{\prod}$ can be written as $\mathcal{L}^{\mathsf{add}}_{\prod}(f, \mathsf{DB}(f)) = \overline{\mathcal{L}}(f, |\mathsf{DB}(f)|)$ or $\mathcal{L}^{\mathsf{add}}_{\prod}(w, f) = \overline{\mathcal{L}}(f)$.*

Definition 5 (Weak backward privacy)**.** *An $\mathcal{L}_{\prod}$-adaptive-secure DSSE scheme achieves weak backward privacy if there exist stateless leakage functions $\overline{\mathcal{L}}$, $\overline{\mathcal{L}}'$ and $\widetilde{\mathcal{L}}$ s.t the update and search leakages can be written as $\mathcal{L}^{\mathsf{add}}_{\prod}(w, f) = \overline{\mathcal{L}}(w)$, $\mathcal{L}^{\mathsf{del}}_{\prod}(w, f) = \overline{\mathcal{L}}'(w)$ and $\mathcal{L}^{\mathsf{srh}}_{\prod}(w) = \widetilde{\mathcal{L}}(\mathsf{TimeDB}(w), \mathsf{DelHist}(w))$.*

5.3 Security of FLY

Theorem 1. *If Z, F, G, P are secure PRFs, then FLY is $\mathcal{L}_{\prod}$-adaptively-secure in the random oracle model, where the leakage functions $\mathcal{L}_{\prod}$ are defined as $\mathcal{L}^{\mathsf{set}}_{\prod}(\lambda) = \emptyset$, $\mathcal{L}^{\mathsf{add}}_{\prod}(f, \mathsf{DB}(f)) = (f, |\mathsf{DB}(f)|)$, $\mathcal{L}^{\mathsf{del}}_{\prod}(f) = f$, and $\mathcal{L}^{\mathsf{srh}}_{\prod}(w) = (\mathsf{SP}(w), \mathsf{UpHist}(w))$.*

Note that FLY satisfies forward privacy since $\mathcal{L}^{\mathsf{add}}_{\prod}$ leaks only $(f, |\mathsf{DB}(f)|)$, and $\overline{\mathcal{L}}$ can be defined as $\mathcal{L}^{\mathsf{add}}_{\prod}$. The proof can be found in Appendix C.

5.4 Security of FLY+ and FLY++

Theorem 2. *If Z, F, G, P are secure PRFs, and SPE is semantically secure, FLY+ is $\mathcal{L}_{\prod}$-adaptively-secure in the random oracle model. The leakage functions are defined as $\mathcal{L}^{\mathsf{set}}_{\prod}(\lambda) = \emptyset$, $\mathcal{L}^{\mathsf{srh}}_{\prod}(w) = (\mathsf{SP}(w), \mathsf{TimeDB}(w), \mathsf{DelHist}(w))$, $\mathcal{L}^{\mathsf{add}}_{\prod}(w, f) = \emptyset$, and $\mathcal{L}^{\mathsf{del}}_{\prod}(w, f) = \mathsf{DelTime}$.*

$\mathsf{DelTime} = \{(v^{\mathbf{add}}, v^{\mathbf{del}}) | \exists (w, f) s.t. (v^{\mathbf{add}}, \mathbf{add}, (w, f)) \in Q \wedge (v^{\mathbf{del}}, \mathbf{del}, (w, f)) \in Q\}$. The definition of DelTime is similar to that of $\mathsf{DelHist}(w)$, except that DelTime has no connection with keyword. FLY+ achieves forward and backward privacy. The proof can be found in Appendix D.

The security analysis of FLY++ (sketch). Compared with FLY+, FLY++ leaks which pairs belong to the same file in the update phase. This extra leakage is about the deletion lists, and has no connection with the newly added keywords. Therefore, the forward privacy of FLY++ is straightforward because the leakage in the addition phase is the same as FLY+. As for backward privacy, each file identifier is encrypted with SE under a symmetric key k_e, which is a combination of a primary key recoverable by the CS and a secondary key recoverable by the AC. Due to the semantic security of SE, the CS cannot recover the file identifier without k_e. For pair-based deletion, the CS cannot regenerate the primary key due to the security of SPE. For file-based deletion, the AC cannot regenerate the secondary key due to the security of PPRFs. As longs as there is no collusion between the CS and the AC, the symmetric keys of deleted keyword/file pairs cannot be recovered.

6 Evaluation

In this section, we provide performance analysis and evaluation for the proposed schemes. To validate their effectiveness, we compare FLY with the state-of-the-art FP schemes, Sophos [16], FAST [18] and Dual [21], and we compare FLY+ with Janus++ [28], the most efficient single-round BP scheme. As for FLY++, its construction is similar to that of FLY+, hence we mainly show its difference between FLY+.

6.1 Performance Analysis

We theoretically analyze the performance of all schemes in terms of data size and communication/computational complexities, and provide the comparison results in Table 2. We start with four FP schemes, where Sophos and FAST implement deletion operations through addition operations, thus failing to support actual deletion. As for data size, all schemes have similar complexities except that FLY requires the CS to maintain an extra deletion map, and Sophos and FAST require the CS to maintain the keyword/file pairs that have been deleted in addition. As for searching a keyword w, all schemes send a search token of constant size to the CS, which returns matched results, thereby having the same communication complexity (In FLY, the size of search certificates generated by the DO is the same as that of search tokens issued by the AC). The main difference lies in the computational complexity at the CS side: FLY checks only the added files containing keyword w, incurring the most optimal search cost; Sophos and FAST require all deleted files containing keyword w to be examined in addition; Dual requires the search results for keyword w to be re-encrypted at the end of the search phase. To update a file with k keywords, all schemes incur similar complexities except that Dual and FLY, with support for file-based deletion, incur constant time to generate a constant-size deletion token.

TABLE 2
Comparison with Prior Work.

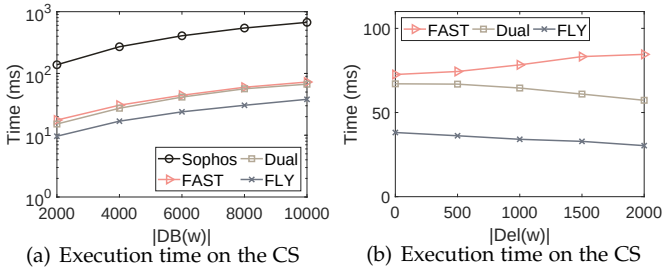
Scheme	Data size		Communication				Computation						FP	BP
			Search		Addition	Deletion	Search		Addition		Deletion			
	DO	CS	DO	CS	Token	Token	DO	CS	DO	CS	DO	CS		
Sophos	$O(m)$	$O(N^+)$	$O(1)$	$O(n_w)$	$O(k)$	$O(k)$	$O(1)$	$O(u_w)$	$O(k)$	$O(k)$	$O(k)$	$O(k)$	✓	-
FAST	$O(m)$	$O(N^+)$	$O(1)$	$O(n_w)$	$O(k)$	$O(k)$	$O(1)$	$O(u_w)$	$O(k)$	$O(k)$	$O(k)$	$O(k)$	✓	-
Dual	$O(m)$	$O(N)$	$O(1)$	$O(n_w)$	$O(k)$	$O(1)$	$O(1)$	$O(a_w + n_w)$	$O(k)$	$O(k)$	$O(1)$	$O(k)$	✓	-
FLY	$O(m)$	$O(N + n)$	$O(1)$	$O(n_w)$	$O(k)$	$O(1)$	$O(1)$	$O(a_w)$	$O(k)$	$O(k)$	$O(1)$	$O(k)$	✓	-
Janus++	$O(m)$	$O(N^+)$	$O(1)$	$O(n_w)$	$O(k)$	$O(k)$	$O(1)$	$O(u_w + n_w)$	$O(k)$	$O(k)$	$O(k)$	$O(k)$	✓	✓
FLY+	$O(m)$	$O(N^+)$	$O(1)$	$O(n_w)$	$O(k)$	$O(k)$	$O(1)$	$O(a_w)$	$O(k)$	$O(k)$	$O(k)$	$O(k)$	✓	✓
FLY++	$O(m + n)$	$O(N^+)$	$O(1)$	$O(n_w)$	$O(k)$	$O(1)$	$O(1)$	$O(a_w)$	$O(k)$	$O(k)$	$O(1)$	$O(k)$	✓	✓

The update complexity is based on adding/deleting a file containing k keywords. For keyword w, a_w is the number of added files including w, d_w is the number of deleted files including w, $u_w = a_w + d_w$ is the number of updates for keyword w, and $n_w = a_w - d_w$ is the number of search results under keyword w. N is the total number of keyword/file pairs in the database, $N^+ = \sum_w (a_w + d_w)$ is the number of keyword/file pairs historically stored in the database, m is the number of distinct keywords, and n is the number of files in the database.

TABLE 3
Performance of FP Schemes in Setup and Update.

Scheme	Setup			Addition	Deletion
	Time(s)	Size1(MB)	Size2(MB)	Time(μs)	Time(μs)
Sophos	3469	31.1	200	1501	1501
FAST	30.23	2.96	297	13.08	13.09
Dual	22.23	5.38	502	9.62	0.68
FLY	34.06	5.15	649	14.74	2.07

Size1: the storage cost at DO side, Size2: the storage cost at CS side.



(a) Execution time on the CS (b) Execution time on the CS

Fig. 5. Comparison of FP schemes in the search phase. (a) The search time for querying a keyword included in $|\mathsf{DB}(w)|$ files with fixed number of deletions $|\mathsf{Del}(w)| = 0$. (b) The search time after deleting $|\mathsf{Del}(w)|$ files with fixed number of matched files $|\mathsf{DB}(w)| = 10,000$.

For the BP schemes, SPE instantiated by GGM trees [32] is the cryptographic primitive. The performance of SPE is affected by the number of GGM trees determined the maximal number of deletions d, and the height of GGM trees t. In empirical study, the values of d and t are relatively small, Hence, we treat d and t as constants in the following analyses, and demonstrate their effects by experiments. As for the data size, all schemes requires the CS to store the public key shares for the deleted keyword/file pairs (with complexity $O(\sum_w d_w)$) besides the added keyword/file pairs (with complexity $O(\sum_w a_w)$). Compared with Janus++ and FLY+, FLY++ requires the DO to extra store a deletion map (with complexity $O(n)$) to generate a constant-size deletion token. To search a keyword w, Janus++ requires access to two FP instances, which include $O(a_w)$ and $O(d_w)$ elements relevant to keyword w, respectively; FLY+ and FLY++ performs searches only on the hybrid list HL_w including $O(a_w)$ elements. Besides, the original Janus++ explicitly caches all the search results in the clear forms. In order to realize authorized searches, Janus++ needs to re-encrypt the search results, incurring the extra overhead $O(n_w)$.

6.2 Experiment Settings and Dataset

In the experiments, a server with an Intel(R) Xeon(R) Gold 5218 2.3GHz CPU and 128GB memory is regard as the CS, a laptop equipped with Intel Core i5 1.6GHz CPU is used to run the DO-side program, and a smartphone equipped with HUAWEI Kirin 990 5G SoC is used to run the AC-side program. The code is written in Java and an Android application is developed and installed on the smartphone. The security parameter λ is set to 256 bits. As for cryptographic algorithms, HMAC is used for PRFs, and SHA-256 and SHA-512 are used to implement the keyed hash functions. In addition, the symmetric encryption algorithm in Janus++ and FLY++ is implemented with AES-256, and the GGM tree in SPE and PPRF is implemented via HMAC.

We conduct experiments on a well-known real dataset, Enron Emai[3], which contains $577,744$ plaintext files. Before experiments, the dataset is preprocessed by extracting $1\sim10$ keywords from each file with the TextRank algorithm and TF-IDF model. The extracted keyword count is $54,021$, and the total number of keyword/file pairs is $2,310,976$.

6.3 Performance Comparison for FLY

Setup. Table 3 describes the data size and the DO-side execution time for constructing the whole dataset. From Table 3, we know that Sophos consumes the most execution time because of the adoption of asymmetric encryption. In terms of the data size on the CS side, Dual and FLY are more expensive compared with Sophos and FAST. This is because both Dual and FLY incorporate inverted indexes and forward indexes to support actual deletion. Furthermore, FLY consumes the largest CS-side storage cost, since FLY disperses the previous secret keys into the Hybrid index to avoid the re-encryption of search results while maintaining a deletion map for file-based deletion.

Update. Table 3 also displays the execution time at the DO side for updating a single keyword/file pair. The results of addition operations are consistent with those in setup phase. For Sophos and FAST, the execution time of deleting one pair is almost the same as the addition time. The reason

3. http://www.cs.cmu.edu/∼./enron/

TABLE 4
Performance of BP Schemes in Setup and Update.

Scheme	t	d	Setup			Addition	Deletion
			Time(s)	Size1(MB)	Size2(MB)	Time(μs)	Time(μs)
Janus++	16	10	879	5.58	196	380	346
	16	30	2441	5.58	196	1056	346
	32	10	1745	5.58	231	755	1199
	32	30	4859	5.58	231	2103	1199
FLY+	16	10	955	7.73	553	413	368
	16	30	2531	7.73	553	1095	368
	32	10	1821	7.73	588	788	1275
	32	30	4953	7.73	588	2143	1275
FLY++	16	10	1009	26.46	624	437	0.75
	16	30	2585	26.46	624	1118	0.75
	32	10	1875	26.46	659	811	0.75
	32	30	5007	26.46	659	2167	0.75

Size1: the storage cost at DO side, Size2: the storage cost at CS side.

is that both schemes do not support actual deletion, treating the deletion process as the duplicate of the addition process.

Search. Fig. 5-(a) shows the impact of the number of files containing the searching keyword on the search performance. From this figure, we know that the search time in all four schemes grows linearly with $|\mathsf{DB}(w)|$. Fig. 5-(b) shows the impact of the number of deleted files including the searching keyword on the search performance. Note that Sophos constructed based on asymmetric encryption consumes much more search time compared with other FP schemes, and will be omitted for ease of comparison. This figure shows that FLY and Dual are negatively impacted by the values of $|\mathsf{Del}(w)|$, but FAST is positively impacted by the value of $|\mathsf{Del}(w)|$. Namely, the schemes without support of actual deletion perform worse as the number of deletions increases. For example, the search time of Dual decreases from 67ms to 57ms, but that of FAST increases from 73ms to 85ms as $|\mathsf{Del}(w)|$ increases from 0 to $2,000$.

6.4 Performance Comparison for FLY+ and FLY++

SPE is the cryptographic primitive of Janus++, FLY+ and FLY++. When SPE is instantiated by GGM trees, t denotes the height of GGM trees (t also equals the length of tags) and d as the maximal number of deletions determines the number of GGM trees. In our experiments, we set $t = \{16, 32\}$ and d ranging from 10 to 30. Furthermore, Janus++ uses two FP instances Σ_{add} and Σ_{del} to store keyword/file pairs and public key shares, respectively. To implement Σ_{add} and Σ_{del}, we encrypt each newly inserted data with a fresh key and a counter as the solution adopted in Dual. In order to achieve authorized searches, Janus++ re-encrypts the search results with AES-256 at the end of search queries.

Setup. From Table 4, we know that Janus++ performs best and FLY++ performs worst for creating the whole EDB. As for the DO-side storage, FLY+ additionally maintains the location and key for each keyword compared to Janus++, and FLY++ extra stores a deletion map for file-based deletion compared to FLY+. As for the CS-side storage, FLY+ additionally keeps the location and key of the successor node in each node compared to Janus++, and FLY++ keeps an extra file counter in each node compared to FLY+. Due to the same reason, FLY++ requires the DO to generate and encrypt the

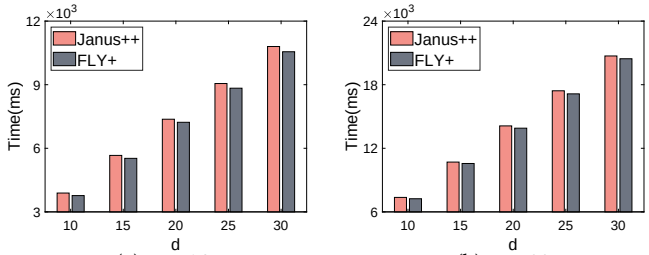


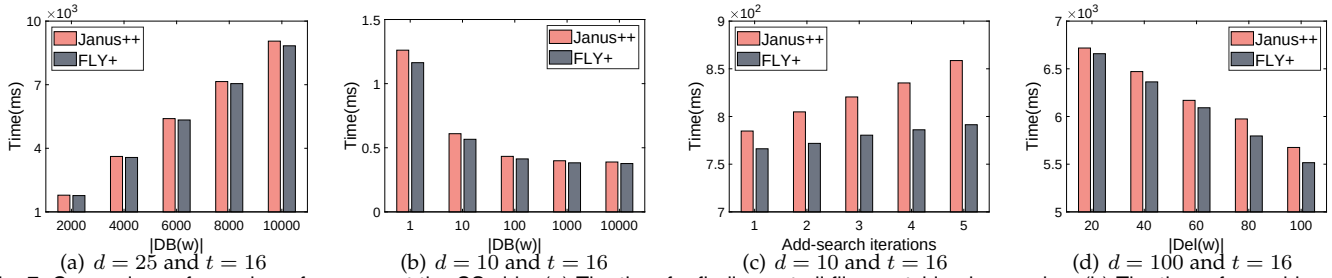
Fig. 6. Comparison of search performance at CS side. The time for searching $10,000$ files without deletion.

extra information, incurring the worst-case execution time. We also observe that the values of d and t have a positive impact on the DO-side execution time. The reason is the more the number of trees or the higher the height of trees, the more the number of HMAC calls. Besides, the CS-side data size of all schemes increases with the increase of t.

Update. Table 4 also displays the execution time for updating each keyword/file pair. Since the pair-based deletion of FLY++ is the same as that in FLY+, we just illustrate the time of file-based deletion for FLY++ in this table. From Table 4, we can get that the results of adding one pair are accord with those in the setup phase. As for deleting a keyword/file pair, the execution time of Janus++ and FLY+ is linear with the tag length t (i.e., the tree height). In SPE, each tag corresponds to a leaf node of a GGM tree, and the deletion of a keyword/pair requires puncturing its tag on one GGM tree. To puncture the tag, the DO needs to compute HMACs for all the siblings on the path from the root to the corresponding leaf node. Therefore, the higher the height of GGM trees, the more the number of HMACs calls. On the other hand, the puncture operation is only performed on one GGM tree regardless of the number of trees. As for FLY++, the deletion map is used to delete all pairs at once. For fair comparison, the time of deleting a pair in FLY++ is calculated by dividing the time of deleting a file by the number of pairs in this file. The results show that the deletion time in FLY++ is constant and unaffected by d or t.

Search. The search performance of FLY++ is the same as FLY+ at the CS side, hence we only illustrate the performance comparisons between FLY+ and Janus++ in Fig. 6 and Fig. 7. From Fig. 6, we know that the search time of both schemes is positively impacted by the values of d and t. In addition, FLY+ is faster than Janus++ under the same settings. The main reason is that Janus++ needs to re-encrypt the search results at the end of search queries.

For more comprehensive comparison, we evaluate the search latency of Janus++ and FLY+ in various conditions. The results in Fig. 7 show that FLY+ always incurs less search time compared with Janus++. The main reason is that our dual-key mechanism avoids the re-encryption of search results. Fig. 7-(a) shows the impact of the number of files containing the searching keyword on the search performance. From this figure, we know that the time for finding all matched files grows linearly with $|\mathsf{DB}(w)|$ in both schemes. From Fig. 7-(b), we can see that the time for finding per file decreases as $|\mathsf{DB}(w)|$ increases in both schemes. The reason is that the initialization cost in the search process can



(a) $d = 25$ and $t = 16$ (b) $d = 10$ and $t = 16$ (c) $d = 10$ and $t = 16$ (d) $d = 100$ and $t = 16$

Fig. 7. Comparison of search performance at the CS side. (a) The time for finding out all files matching keyword w. (b) The time of searching per file matching keyword w. (c) The time for finding out all files matching keyword w after multiple add-search iterations. (d) The time for finding out 10,000 matched files for keyword w after deleting $|\mathsf{Del}(w)|$ files.

TABLE 5
The Storage Cost at AC Side (KB).

Scheme	$\|\Omega\|$=2000	$\|\Omega\|$=4000	$\|\Omega\|$=6000	$\|\Omega\|$=8000	$\|\Omega\|$=10000
FLY	195	390	586	781	976
FLY+	293	586	879	1172	1465
FLY++	356	711	1067	1422	1778

TABLE 6
The Computation Cost of FLY++ at AC Side (ms).

$\|\mathsf{DB}(w)\|$	2000	4000	6000	8000	10000
$t = 20$	405	784	1154	1564	1919
$t = 30$	524	1052	1592	2154	2646
$t = 40$	673	1357	2049	2766	3406

be amortized into more search queries as $|\mathsf{DB}(w)|$ increases. Fig. 7-(c) shows the impact of the dual-key mechanism on the search performance. Specifically, we insert $2,000$ files containing a keyword into the database five times, and issue a search query after each insertion interval. In Fig. 7-(c), we find that the search time difference between FLY+ and Janus++ is getting larger and larger as the number of add-search iterations increases. This is because Janus++ needs to perform symmetric decryption to recover the old search results cached at the CS side in addition to re-encrypting all the new search results. To better show the impact of the number of deleted files, $\mathsf{Del}(w)|$, on search performance, we specially set the maximum value of d to 100 and illustrate the results in Fig. 7-(d). From this figure, we know that the search time of both schemes decreases as $|\mathsf{Del}(w)|$ increases.

6.5 Search Performance on Mobile Devices

The search performance on mobile devices is evaluated in terms of AC-side storage cost and the AC-side execution time. For all our schemes, the AC needs to locally maintain a search certificates Ω so as to generate search tokens. As for the search time, the AC in FLY and FLY+ directly retrieves the plaintext file identifiers from the CS, consuming negligible costs, and in contrast, the AC in FLY++ needs to recover search results by itself, thereby incurring extra costs.

Storage cost. The number of authorized keywords, $|\Omega|$, is set in a range from $2,000$ to $10,000$, and the experiments results are illustrated in Table 5. From this table, we find that the storage costs of all schemes grow as $|\Omega|$ increases. The reason is oblivious, the more authorized keywords means the more keys needed to be generated. Under the same $|\Omega|$, the storage cost of FLY is the smallest, and that of FLY++ is the largest. This is because, for each keyword, FLY++ requires the AC to store extra punctured key k_{S} compared with FLY+, which in turn requires extra authorization counter and local key share compared with FLY. In addition, we can observe that the storage cost on the smartphone is very small. For instance, the size of search certificates containing $10,000$ keywords in FLY++ is less than 2MB.

Search time. To recover search results, the AC needs to perform decryption and evaluate PPRFs in FLY++.Search. The costs of both operations are mainly impacted by the number of files containing the keyword $|\mathsf{DB}(w)|$ and the height of GGM tree t. Table 6 shows the AC-side execution time in FLY++ under different setting of $|\mathsf{DB}(w)|$ and t. From this table, we can find the execution time is positively impacted by the values of $|\mathsf{DB}(w)|$ and t. The reason are two-fold: (1) The larger $|\mathsf{DB}(w)|$ means more file identifiers need to be decrypted. (2) The larger t means more HMAC calls are required in algorithm $\widetilde{\mathsf{F}}.\mathsf{Eval}$. Besides, the results also demonstrate that FLY++ is lightweight and is suitable for resource-limited smartphones. For example, when $t = 20$, it just needs 1.9s for the AC to recover $10,000$ file identifiers.

7 Related Work

This paper devises DSSE schemes with forward and backward privacy to achieve authorized search, flexible update, lightweight, and secure deletion. Therefore, we will introduce the related work in terms of FP DSSE and BP DSSE.

DSSE schemes. The first DSSE scheme with support for sublinear search time was proposed by Kamara et al. [6]. Their subsequent work [7] supported parallel search based on the red-black tree while leaking less information. A DSSE scheme was proposed by Cash et al. [8] for a super-large dataset, but it only supported efficient file insertion. Naveed et al. [9] put forward a DSSE scheme in which the cloud server worked as a blind storage to decrease leakage at the cost of multiple rounds of interaction. Hahn et al. [10] utilized both inverted and forward indexes to achieve efficient updates. Our previous work [11] designed a ranked inverted index and a verifiable matrix on a dynamic document collection to enable verifiable top-K searches. However, the above schemes lack both forward and backward privacy.

FP DSSE schemes. The first FP construction was proposed in 2005 by Chang et al. [14], but no solution with sublinear search time existed before the work of Stefanov et al. [15]. However, the work in [15] is based on oblivious RAM, which produces non-negligible communication cost. Bost et al. [16] formally defined forward privacy and put forward an FP scheme named Sophos, based on asymmetric cryptography. After that, Wang et al. [17] proposed

FEncKV which applied trapdoor permutation functions in IoT-enabled healthcare systems to realize forward privacy. Considering the high computation cost of asymmetric cryptography, Song et al. [18] constructed two FP schemes FAST and FASTIO by using only symmetric primitives. In addition, Wei et al. [19] constructed an FP scheme FSSE based on a key-based blocks chain. He et al. [20] designed a two-level fish-bone chain to derive an FP scheme with constant local storage. However, these FP schemes [16]–[20] failed to support actual deletion. The first FP DSSE scheme with support for actual deletion was proposed by Kim et al. [21]. We denote this scheme with Dual since it utilizes a dual dictionary to simultaneously achieve sublinear search time and efficient updates. Specifically, Dual uses a fresh key to encrypt the newly added data for forward privacy. To avoid keeping all these keys locally, Dual re-encrypts search results with a new key and discard old keys after each search query. Therefore, the more number of files being searched, the worse the search performance for Dual.

BP DSSE schemes. Bost et al. [22] formally defined backward privacy at three levels: Type-I, Type-II, and Type-III, ordered from most to least secure. To achieve Type-I backward privacy, existing schemes either adopted expensive primitives or required trusted hardware. For example, Moneta [22] and Orion [23] were constructed based on obvious RAM, and Amjad et al. [24] constructed strong BP schemes by utilizing the power of SGX. Type-II backward privacy can be achieved from FP schemes. For example, Fides [22] was obtained from Sophos [16] by storing file/operation pairs encrypted under SE. Specifically, the search client decrypts the set of matched pairs, locally removes the deleted ones, and requests the remaining files from the cloud server. Mitra [23] adopted a similar idea, but utilized only symmetric primitive. Demertzis et al. [25] proposed two Type-II schemes, SD_a and SD_d, to further reduce local storage. However, both schemes required an extra round of interaction in the search phase. To overcome this drawback, Sun et al. [26] proposed a generic construction based on a symmetric revokable encryption scheme.

Type-III (weak) backward privacy can be achieved by different kinds of primitives. Horus [23] utilized oblivious RAM and thus incurred expensive overheads. $\mathsf{Diana}_\mathsf{del}$ [22] was constructed based on constrained pseudorandom functions [33]. Khons [27] developed the hidden pointer technique with support for actual deletion to achieve both forward and backward privacy. However, both schemes required two-round interaction in the search phase. The first non-interactive Type-III BP scheme, named Janus [22], employed public puncturable encryption (PPE) [34] to revoke the cloud server's searching ability on deleted keyword/file pairs. Specifically, in Janus, each keyword/file pair is associated with a tag, and the secret key is punctured on a set of tags so that the ciphertexts associated with these tags cannot be decrypted by the cloud server. However, Janus can be hardly deployed in practice due to the expensive primitive PPE. As an improvement, Sun et al. [28] presented SPE and instantiated a practical BP scheme, Janus++ to support secure pair-based deletions. To avoid re-encryption, Janus++ caches the search results in clear forms, and thus is hard to support authorized keyword searches.

8 Conclusion

In this paper, we design DSSE schemes based on a flexible index Hybrid, so as to achieve authorized keyword search on mobile devices while supporting secure and flexible updates. We first present an efficient FP scheme FLY by encrypting the newly added data with a fresh key while dispersing previous keys into Hybrid. On this basis, we provide a BP scheme FLY+ by further employing SPE and the dual-key mechanism to achieve all our design goals except for flexibility. Finally, we puncture the dual-key mechanism by using SPE and PPRFs to obtain the full-fledged scheme FLY++. Experimental results demonstrate that our schemes are efficient, and outperform the state-of-the-art schemes in the search phase. As part of our future work, we will try to realize a single-round BP scheme with support for flexible updates, while extending our threat model to allow the collusion between the CS and AC.

Acknowledgment

This work was supported in part by the NSFC grants 62272150, 61872133, U20A20181, and 62272162; the Hunan Provincial Natural Science Foundation of China (Grant No. 2020JJ3015); and the Postgraduate Scientific Research Innovation Project of Hunan Province (No. QL20210095).

References

[1] Q. Tong, Y. Miao, H. Li, X. Liu, and R. Deng, "Privacy-preserving ranked spatial keyword query in mobile cloud-assisted fog computing," *IEEE Transactions on Mobile Computing*, 2021.
[2] Z. Ma, S. Zhang, J. Liu, X. Liu, W. Wang, J. Wang, and S. Guo, "RF-Siamese: approaching accurate rfid gesture recognition with one sample," *IEEE Transactions on Mobile Computing*, 2022.
[3] C. Cimpanu, "Intel investigating breach after 20GB of internal documents leak online," [Online]. Available: https://www.zdnet.com/article/intel-investigating-breach-after-20gb-of-internal-documents-leak-online
[4] Y. Lu, and J. Li, "Lightweight public key authenticated encryption with keyword search against adaptively-chosen-targets adversaries for mobile devices," *IEEE Transactions on Mobile Computing*, 2022.
[5] Q. Liu, Y. Peng, J. Wu, T. Wang, and G. Wang, "SlimBox: lightweight packet inspection over encrypted traffic," *IEEE Transactions on Dependable and Secure Computing*, 2022.
[6] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. of CCS*, 2012.
[7] S. Kamara, and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Proc. of FC*, 2013.
[8] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: data structures and implementation," in *Proc. of NDSS*, 2014.
[9] M. Naveed, M. Prabhakaran, and C. A. Gunter, "Dynamic searchable encryption via blind storage," in *Proc. of S&P*, 2014.
[10] F. Hahn, and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *Proc. of CCS*, 2014.
[11] Q. Liu, Y. Tian, J. Wu, T. Peng , and G. Wang, "Enabling verifiable and dynamic ranked search over outsourced data," *IEEE Transactions on Services Computing*, 2022.
[12] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. of S&P*, 2000.
[13] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proc. of CCS*, 2006.
[14] Y.-C. Chang, and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Proc. of ACNS*, 2005.
[15] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. of NDSS*, 2014.

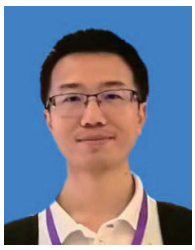
[16] R. Bost, "Σοφος: Forward secure searchable encryption," in *Proc. of CCS*, 2016.
[17] K. Wang, C.-M. Chen, Z. Tie, M. Shojafar, S. Kumar, and S. Kumari, "Forward privacy preservation in IoT-enabled healthcare systems," *IEEE Transactions on Industrial Informatics*, 2022.
[18] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward private searchable symmetric encryption with optimized I/O efficiency," *IEEE Transactions on Dependable and Secure Computing*, 2020.
[19] Y. Wei, S. Lv, X. Guo, Z. Liu, Y .Huang, and B .Li, "FSSE: Forward secure searchable encryption with keyed-block chains," *Information Sciences*, 2019.
[20] K. He, J. Chen, Q. Zhou, R. Du, and Y. Xiang, "Secure dynamic searchable symmetric encryption with constant client storage cost," *IEEE Transactions on Information Forensics and Security*, 2021.
[21] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proc. of CCS*, 2017.
[22] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. of CCS*, 2017.
[23] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *Proc. of CCS*, 2018.
[24] G. Amjad, S. Kamara, and T. Moataz, "Forward and backward private searchable encryption with SGX," in *Proc. of EuroSec*, 2019.
[25] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," in *Proc. of NDSS*, 2020.
[26] S.-F. Sun, R. Steinfeld, S. Lai, X. Yuan, A. Sakzad, J. Liu, S. Nepal, and D. Gu, "Practical non-interactive searchable encryption with forward and backward privacy," in *Proc. of NDSS*, 2021.
[27] J. Li, Y. Huang, Y. Wei and S. Lv, Z. Liu, C. Dong, and W. Lou, "Searchable symmetric encryption with forward search privacy," *IEEE Transactions on Dependable and Secure Computing*, 2021.
[28] S.-F. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *Proc. of CCS*, 2018.
[29] Q. Liu, Y. Peng, J. Wu, T. Wang , and G. Wang, "Secure multi-keyword fuzzy searches with enhanced service quality in cloud computin," *IEEE Transactions on Network and Service Management*, 2021.
[30] S. Hohenberger, V. Koppula, and B. Waters, "Adaptively secure puncturable pseudorandom functions in the standard model," in *Proc. of ASIACRYPT*, 2015.
[31] Q. Liu, G. Wang, and J. Wu, "Time-based proxy re-encryption scheme for secure data sharing in a cloud environment," *Information Sciences*, 2014.
[32] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions (extended abstract)," in *Proc. of FOCS*, 1984.
[33] D. Boneh, and B. Waters, "Constrained pseudorandom functions and their applications," in *Proc. of ASIACRYPT*, 2013.
[34] M. D. Green, and I. Miers, "Forward secure asynchronous messaging from puncturable encryption," in *Proc. of S&P*, 2015.

Qin Liu received her B.Sc. in Computer Science in 2004 from Hunan Normal University, China, received her M.Sc. in Computer Science in 2007, and received her Ph.D. in Computer Science in 2012 from Central South University, China. She has been a Visiting Student at Temple University, USA. Her research interests include security and privacy issues in cloud computing. Now, she is an Associate Professor in the College of Computer Science and Electronic Engineering at Hunan University, China.

Yu Peng is currently working toward the PhD degree with the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include the security and privacy issues in cloud computing, networked applications and blockchain.

Hongbo Jiang received the PhD degree from Case Western Reserve University, in 2008. After that, he joined the faculty of the Huazhong University of Science and Technology as a full professor and the dean of the Department of Communication Engineering. Now, he is a full professor with the College of Computer Science and Electronic Engineering, Hunan University. His research concerns computer networking, especially algorithms and protocols for wireless and mobile networks.He is serving as an editor for the IEEE/ACM Transactions on Networking, associate editor for the IEEE Transactions on Mobile Computing, and associate technical editor for the IEEE Communications Magazine.

Jie Wu is the Chair and a Laura H. Carnell Professor in the Department of Computer and Information Sciences at Temple University, Philadelphia, PA, USA. Prior to joining Temple University, he was a Program Director at the National Science Foundation and a Distinguished Professor at Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu has regularly published in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE TRANSACTIONS ON SERVICE COMPUTING, and Journal of Parallel and Distributed Computing. Dr. Wu was general co-chair/chair for IEEE MASS 2006, IEEE IPDPS 2008, IEEE ICDCS 2013, and ACM MobiHoc 2014, as well as program co-chair for IEEE INFOCOM 2011 and CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.



Tian Wang received his BSc and MSc degrees in Computer Science from the Central South University in 2004 and 2007, respectively. He received his PhD degree in City University of Hong Kong in 2011. Currently, he is a professor at the Institute of Artificial Intelligence and Future Networks, Beijing Normal University & UIC, China. His research interests include internet of things and edge computing.

Tao Peng received the B.Sc. in Computer Science from Xiangtan University, China, in 2004, the M.Sc. in Circuits and Systems from Hunan Normal University, China, in 2007, and the Ph.D. in Computer Science from Central South University, China, in 2017. Now, she is an Associate Professor of School of Computer Science and Cyber Engineering, Guangzhou University, China. Her research interests include network and information security issues.

Guojun Wang received B.Sc. degree in Geophysics, M.Sc. degree in Computer Science, and Ph.D. degree in Computer Science, at Central South University, China, in 1992, 1996, 2002, respectively. He is a Pearl River Scholarship Distinguished Professor of Higher Education in Guangdong Province, a Doctoral Supervisor of School of Computer Science and Cyber Engineering, Guangzhou University, China, and the Director of Institute of Computer Networks at Guangzhou University. He has been listed in Chinese Most Cited Researchers (Computer Science) by Elsevier in the past eight consecutive years (2014-2021). His research interests include artificial intelligence, big data, cloud computing, Internet of Things (IoT), blockchain, trustworthy/dependable computing, network security, privacy preserving, recommendation systems, and smart cities. He is a Distinguished Member of CCF, a Member of IEEE, ACM and IEICE.

APPENDIX A
ILLUSTRATIVE EXAMPLE OF SPE.

In this section, we will give an example to illustrate the high-level idea of SPE. Suppose the database consists of four keyword/file pairs $\{(w_1, f_1), (w_1, f_2), (w_1, f_3), (w_1, f_4)\}$. If the maximal number of deletions is set to $d = 1$, for keyword w_1, the process of SPE can be instantiated by two GGM trees with height 2 as shown in Fig. 1. In the GGM tree, each node $\mathcal{N}_i$ is associated with a binary tag τ_i and a key k_{τ_i}, in which the key can be derived from that of its ancestor node but not vice versa. The root key of a GGM tree equals the local key share, and each leaf node corresponds to a keyword/file pair. To add a pair with tag τ_i, algorithm $\mathsf{SPE.GenKey}(\mathsf{lks}^{(w_1)}, \tau_i)$ is run to generate the encryption key $\mathsf{key}^{(\tau_i)}$, which is a combination of the keys of leaf nodes with tag τ_i in two GGM trees, i.e., $\mathsf{key}^{(\tau_i)} = k_{\tau_i} \oplus k'_{\tau_i}$. If a pair with tag τ_i is deleted, algorithm $\mathsf{SPE.IncPun}(\mathsf{lks}^{(w_1)}, \tau_i)$ is run to puncture the leaf node with tag τ_i in one GGM, making $\mathsf{key}^{(\tau_i)}$ unrecoverable by algorithm SPE.RegenKey.

For example, for pair (w_1, f_1) with tag τ_1, $\mathsf{key}^{(\tau_1)}$ is in the form of $k_{\tau_1} \oplus k'_{\tau_1}$. Suppose that (w_1, f_1) is the first deleted pair. SPE punctures tag τ_1 on the first GGM tree to generate the public key share $\mathsf{pks}^{(\tau_1)}$ (the keys of red nodes in the first GGM tree) and the new local key share $\mathsf{lks}'^{(w_1)}$ (the root key of the second GGM tree). In this way, the user with $\mathsf{pks}^{(\tau_1)}$ and $\mathsf{lks}'^{(w_1)}$ can just regenerate the keys for the undeleted pairs as follows: For $i \in \{2, 3, 4\}$, it first recovers keys k_{τ_i} and k'_{τ_i} by running $\widetilde{\mathsf{F}}.\mathsf{Eval}(\mathsf{pks}^{(\tau_i)}, \tau_i)$ and $\widetilde{F}(\mathsf{lks}'^{(w)}, \tau_i)$, respectively, and then it uses the key $\mathsf{key}^{(\tau_i)} = k_{\tau_i} \oplus k'_{\tau_i}$ to recover the encrypted pair with tag τ_i. As for (w_1, f_1), the user does not have the key k_{τ_1} and thus cannot recover $\mathsf{key}^{(\tau_1)}$ due to the security of hash functions. Once a query is performed on keyword w_1, the DO needs to randomly choose a new local key share for security reasons.

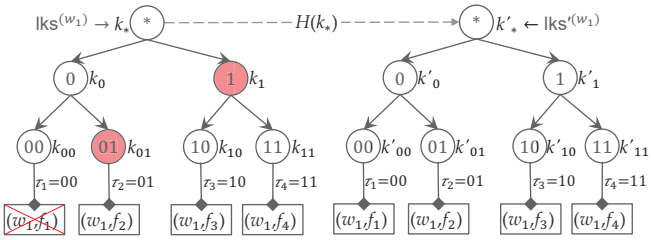


Fig. 1. The working process of SPE. The tag is created by prefix code.

APPENDIX B
TIME-BASED AUTHORIZATION OF FLY++.

In this section, we will provide detailed descriptions of time-based authorization in FLY++. For briefness, we also assume that the time granularity is a day, and the day key $\mathsf{dkey}^{(w)}$ of keyword w can be calculated by the time-based key tree described in Section 3.3. Let $T : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ be a PRF, and let $H_i : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^\lambda (i = 6, 7, 8, 9)$ be keyed hash functions. FLY++ can be extended to reduce authorization interaction as follows:

• Setup. This algorithm is similar to FLY++.Setup, except that the local state σ contains a PRF key $k_0 \in \{0,1\}^\lambda$ in addition, and for each keyword w, the local map $\mathrm{T_C}$ extra stores a keyword counter $c^{(w)}$, which denotes the number of files containing this keyword in current date.

• Addition. At the beginning of each day, the DO increases the authorization counters of all keywords in the local map by 1. Then, the DO uploads the newly collected data and update the local map as follows: For each keyword/file pair (w, f), the DO generates an addition token τ_{add} as FLY++.Addition except that the fresh key $\mathsf{nhead}^{(w)}$ and the new location $\mathsf{nkey}^{(w)}$ are calculated by Eq. (1) in Section 3.3, and $\mathrm{T_C}[w]$ is updated with $(\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}^{(w)}, c^{(w)} + 1)$.

Once all the data have been uploaded, the DO encrypts the key k_{S} by calculating $ek_{\mathsf{S}} \leftarrow \mathsf{SE.Enc}(k_5, k_{\mathsf{S}})$, and for each key $w \in \mathsf{W}$, it generates a label by calculating $l^{(w)} \leftarrow H_9(\mathsf{dkey}^{(w)}, y||m||d)$ and encrypts the keyword counter and relevant keys by the following equations:

$$e^{(w)} \leftarrow \mathsf{SE.Enc}(\mathsf{dkey}^{(w)}, c^{(w)}) \tag{2}$$

$$\mathsf{elks}^{(w)} \leftarrow \mathsf{SE.Enc}(\mathsf{dkey}^{(w)}, \mathsf{lks}^{(w)}) \tag{3}$$

$$\mathsf{epun}^{(w)} \leftarrow \mathsf{SE.Enc}(\mathsf{dkey}^{(w)}, \mathsf{pun}^{(w)}) \tag{4}$$

Next, the DO outsources the extra data $(\{(l^{(w)}, e^{(w)}, \mathsf{elks}^{(w)}, \mathsf{epun}^{(w)})\}_{w \in \mathsf{W}}, ek_{\mathsf{S}})$, and clears all the keyword counters from the local map. In addition, each hybrid node can be associated with a label $\mathsf{label}^{(w)} = Z_{k_{\mathsf{pos}}}(w \| f)$ for flexibility.

• Deletion. As for deleting a file f, the DO performs in the same way as FLY++.Deletion except that the DO uploads the encrypted key $ek_{\mathsf{S}} = \mathsf{SE.Enc}(k_5, k_{\mathsf{S}})$ together with the deletion token. To delete pair (w, f), the DO performs in the same way as FLY++.Deletion except that the DO encrypts the new local key share by Eq. (3) and uploads the ciphertext $\mathsf{elks}^{(w)}$ along with the deletion token.

• Authorization. If the AC is authorized to search keyword w during time period $[s, t]$, the DO first finds the maximal covering nodes MCN in the time-based key tree $\mathcal{T}_w$, and then sets the search certificate as the keys associated with all these nodes, i.e., $\Omega \leftarrow \{\mathcal{N}.key\}_{\mathcal{N} \in \mathsf{MCN}}$.

• Search. To search a keyword w on date (y, m, d), the AC first uses its search certificate Ω to generate the corresponding day key $\mathsf{dkey}^{(w)}$, and sets the search token as $\tau_{srh} = (\mathsf{dkey}^{(w)})$. Given the search token, the CS first locates the ciphertexts $(e^{(w)}, \mathsf{elks}^{(w)}, \mathsf{epun}^{(w)})$ through the label $l^{(w)}$ and then recovers their plaintexts by calculating $c^{(w)} \leftarrow \mathsf{SE.Dec}(\mathsf{dkey}^{(w)}, e^{(w)})$, $\mathsf{lks}^{(w)} \leftarrow \mathsf{SE.Dec}(\mathsf{dkey}^{(w)}, \mathsf{elks}^{(w)})$, and $\mathsf{pun}^{(w)} \leftarrow \mathsf{SE.Dec}(\mathsf{dkey}^{(w)}, \mathsf{epun}^{(w)})$.

Next, the CS calculates the location and key of the current header of hybrid list HL_w by using Eq. (1) in Section 3.3, and performs in the same way as algorithm FLY++.Search to update the hybrid list HL_w and generate the search result IND. At the end of search, the CS sends IND along with the latest ciphertext ek_{S} to the AC, which recovers the key k_{S} by running $\mathsf{SE.Enc}(k_5, ek_{\mathsf{S}})$ and executes lines 12-16 of algorithm FLY++.Search to recover file identifiers.

APPENDIX C
PROOF OF THEOREM 1

Proof: The hash functions are modeled as random oracles. For $i \in [4]$, each oracle maintains a hash table $\mathbb{H}_i$ that stores input/output pairs $(\mathbf{in}, \mathbf{out}) \in \{0,1\}^* \times \{0,1\}^\lambda$.

Algorithm 1 Game G_5

FLY.Setup
1: $(\mathrm{T_C}, \mathrm{T_D}, \mathrm{T_S}) \leftarrow$ empty map
2: $\mathrm{Updates} \leftarrow$ empty map; $v \leftarrow 1$
3: $\sigma \leftarrow \mathrm{T_C}$; $\mathsf{EDB} \leftarrow (\mathrm{T_D}, \mathrm{T_S})$
4: **return** (σ, EDB)

FLY.Addition
1: **if** $\mathrm{FK}[f] = \perp$ **then**
2: $\quad \mathrm{FK}[f] \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^\lambda$
3: $(\mathsf{key}_1^{(f)}, \mathsf{key}_2^{(f)}, \mathsf{key}_3^{(f)}) \leftarrow \mathrm{FK}[f]$
4: $\mathsf{AST} \leftarrow \emptyset$
5: **for** $c \in |\mathsf{DB}(f)|$ **do**
6: $\quad$ append (v, f) to $\mathrm{Updates}[w]$
7: $\quad r \xleftarrow{\$} \{0,1\}^\lambda$; $\mathsf{pos} \xleftarrow{\$} \{0,1\}^\lambda$
8: $\quad \mathrm{RAND}[v] \leftarrow r$; $\mathrm{POS}[v] \leftarrow \mathsf{pos}$
9: $\quad$ **for** $j \in [4]$ **do**
10: $\quad\quad \mathrm{M}_j[v] \xleftarrow{\$} \{0,1\}^\lambda$
11: $\quad$ program H_4 s.t. $\mathbb{H}_4(\mathsf{key}_3^{(f)}, r) \leftarrow \mathrm{M}_4[v]$
12: $\quad$ **if** $c = 1$ **then**
13: $\quad\quad \mathsf{E} \leftarrow (f \oplus \mathrm{M}_1[v], \mathrm{M}_2[v], \mathrm{M}_3[v], \mathrm{M}_4[v])$
14: $\quad$ **else**
15: $\quad\quad \mathsf{E} \leftarrow (f \oplus \mathrm{M}_1[v], \mathrm{M}_2[v], \mathrm{M}_3[v], \mathrm{M}_4[v] \oplus \mathrm{POS}[v-1])$
16: $\quad \mathsf{AST} \leftarrow \mathsf{AST} \cup (\mathsf{pos}, \mathsf{E}, r)$; $v \leftarrow v + 1$
17: **return** $\tau_{\mathsf{add}} = (\mathsf{AST}, \mathsf{key}_1^{(f)}, \mathsf{key}_2^{(f)} \oplus \mathsf{pos})$

FLY.Deletion
1: **return** $\tau_{\mathsf{del}} = \mathrm{FK}[f]$

FLY.Search
1: parse $\mathrm{Updates}[w]$ as $((v_1, f_1), \ldots, (v_T, f_T))$
2: **if** $T = 0 \wedge \mathrm{T_C} = \perp$ **then**
3: $\quad$ **return** $\emptyset$
4: **if** $\mathrm{T_C}[w] = \perp$ **then**
5: $\quad \mathrm{T_C}[w] \leftarrow (\mathbf{0}, \mathbf{0})$
6: **for** each $(v_t, f_t) \in \mathrm{Updates}[w]$ **do**
7: $\quad \mathsf{nhead}^{(w)} \leftarrow \mathrm{POS}[v_t]$; $\mathsf{nkey}^{(w)} \xleftarrow{\$} \{0,1\}^\lambda$
8: $\quad r \leftarrow \mathrm{RAND}[v_t]$; $(\mathsf{head}^{(w)}, \mathsf{key}^{(w)}) \leftarrow \mathrm{T_C}[w]$
9: $\quad$ program H_1 s.t. $\mathbb{H}_1[\mathsf{nkey}^{(w)}, r] \leftarrow \mathrm{M}_1[v_t]$
10: $\quad$ program H_2 s.t. $\mathbb{H}_2[\mathsf{nkey}^{(w)}, r] \leftarrow \mathsf{head}^{(w)} \oplus \mathrm{M}_2[v_t]$
11: $\quad$ program H_3 s.t. $\mathbb{H}_3[\mathsf{nkey}^{(w)}, r] \leftarrow \mathsf{key}^{(w)} \oplus \mathrm{M}_3[v_t]$
12: $\quad \mathrm{T_C}[w] \leftarrow (\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$
13: $(\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)}) \leftarrow \mathrm{T_C}[w]$; $\mathrm{Updates}[w] \leftarrow \perp$
14: **return** $\tau_{\mathsf{srh}} = (\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$

Given an input $\mathbf{in}$, the oracle checks whether $\mathbb{H}_i[\mathbf{in}]$ exists or not. If so, it returns the corresponding entry. Otherwise, it chooses a random string $\mathbf{out}$, stores $(\mathbf{in}, \mathbf{out})$ in $\mathbb{H}_i$, and returns $\mathbf{out}$. The proof proceeds through several games. We start from the real-world game $\mathbf{Real}_{\mathcal{A}}^{\prod}(\lambda)$ and construct a sequence of games that are indistinguishable from their predecessors. Eventually, we reach the last game $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\prod}(\lambda)$. By relating the games, we conclude that $\mathbf{Real}_{\mathcal{A}}^{\prod}(\lambda)$ is indistinguishable from $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\prod}(\lambda)$ and complete our proof.

• **Game** G_0 : This game is exactly the real-world game except for some bookkeeping changes. The experiment maintains a map CNT to record the number of additions for each keyword w. The following codes are added after line 2 of protocol FLY.Addition:

$1:$ **if** $\mathrm{CNT}[w] = \perp$
$2:$ $\quad \mathrm{CNT}[w] \leftarrow 1$
$3:$ **end if**
$4: t \leftarrow \mathrm{CNT}[w]; \mathrm{CNT}[w] \leftarrow t + 1$

Therefore, we have:

$$\Pr[\mathbf{Real}_{\mathcal{A}}^{\prod}(\lambda) = 1] = \Pr[G_0 = 1].$$

• **Game** G_1 : This game is exactly the real-world game except that it replaces the calls to PRFs by picking random strings. The experiment maintains a map FK to store keys $(\mathsf{key}_1^{(f)}, \mathsf{key}_2^{(f)}, \mathsf{key}_3^{(f)})$ for each file f, and uses a map POS to record the position of each keyword/file pair. When $(F_{k_1}(f), G_{k_2}(f), P_{k_3}(f))$ (resp. $Z_{k_{\mathsf{pos}}}(w||f)$) are needed, the experiment checks whether $\mathrm{FK}[f]$ (resp. $\mathrm{POS}[f, c]$) exists or not. If so, it returns the entry; otherwise, it picks random strings and updates the entry accordingly. If the adversary $\mathcal{A}$ can distinguish between $\mathbf{Real}_{\mathcal{A}}^{\prod}(\lambda)$ and G_1, we can build a reduction $\mathcal{B}$ to distinguish PRFs from truly random functions. Therefore, we have:

$$\Pr[G_0 = 1] - \Pr[G_1 = 1] \leq 4 \cdot \mathrm{Adv}_{\mathcal{B}}^{\mathrm{prf}}(\lambda).$$

• **Game** G_2 : This game differs from G_1 in the following aspects. First, the experiment maintains a map WPK to record all the counter/position/key pairs for each keyword w since the last search query. That is, $\mathrm{WPK}[w]$ will be cleared at the end of search phase. The following line is added after line 16 of protocol FLY.Addition:

$1:$ **if** $\mathrm{WPK}[w] = \perp$ **then**
$2:$ $\quad$ **Append** $(t-1, \mathsf{head}^{(w)}, \mathsf{key}^{(w)})$ **to** $\mathrm{WPK}[w]$
$3:$ **Append** $(t, \mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$ **to** $\mathrm{WPK}[w]$

Second, the experiment maintains a map RAND as bookkeeping. We execute the code $\mathrm{RAND}[w, t] \leftarrow r$ after line 10 of protocol FLY.Addition. Third, instead of querying $H_1(\mathsf{nkey}^{(w)}, r)$ to obtain a mask (line 11 in protocol FLY.Addition), the experiment performs as follows:

$$\mathrm{M}_1[w, t] \xleftarrow{\$} \{0,1\}^\lambda; e_1 \leftarrow \mathsf{HN}[1] \oplus \mathrm{M}_1[w, t]$$

where M_1 is a map kept by the experiment. After line 1 of protocol FLY.Search, the following codes are added to program the random oracle H_1.

$1:$ $(t_i, \mathsf{head}_i^{(w)}, \mathsf{key}_i^{(w)})_{i=0}^{T} \leftarrow \mathrm{WPK}[w]$
$2:$ **for** $i = T$ **to** 1 **do**
$3:$ $\quad r \leftarrow \mathrm{RAND}[w, t_i]; \mathsf{nkey}^{(w)} \leftarrow \mathsf{key}_i^{(w)}$
$4:$ $\quad \mathbb{H}_1[\mathsf{nkey}^{(w)}, r] \leftarrow \mathrm{M}_1[w, t_i]$
$5:$ **end for**
$6: \mathrm{WPK}[w] \leftarrow \perp$

G_1 and G_2 behave exactly the same except that in G_2, it is possible for the adversary $\mathcal{A}$ to observe an inconsistency

Simulation 1 Simulator $\mathcal{S}$

Simulation of Setup

1: $(\mathrm{T_C}, \mathrm{T_D}, \mathrm{T_S}) \leftarrow$ empty map; $v \leftarrow 1$
2: $\sigma \leftarrow \mathrm{T_C}$; $\mathsf{EDB} \leftarrow (\mathrm{T_D}, \mathrm{T_S})$
3: **return** (σ, EDB)

Simulation of the addition token

1: **if** $\mathrm{FK}[f] = \perp$ **then**
2: $\quad \mathrm{FK}[f] \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^\lambda$
3: $(\mathsf{key}_1^{(f)}, \mathsf{key}_2^{(f)}, \mathsf{key}_3^{(f)}) \leftarrow \mathrm{FK}[f]$
4: $\mathsf{AST} \leftarrow \emptyset$
5: **for** $c \in |\mathsf{DB}(f)|$ **do**
6: $\quad r \xleftarrow{\$} \{0,1\}^\lambda$; $\mathsf{pos} \xleftarrow{\$} \{0,1\}^\lambda$
7: $\quad \mathrm{RAND}[v] \leftarrow r$; $\mathrm{POS}[v] \leftarrow \mathsf{pos}$
8: $\quad$ **for** $j \in [4]$ **do**
9: $\quad\quad \mathrm{M}_j[v] \xleftarrow{\$} \{0,1\}^\lambda$
10: $\quad$ program H_4, s.t. $\mathbb{H}_4[\mathsf{key}_3^{(f)}, r] \leftarrow \mathrm{M}_4[v]$
11: $\quad$ **if** $c = 1$ **then**
12: $\quad\quad \mathsf{E} \leftarrow (f \oplus \mathrm{M}_1[v], \mathrm{M}_2[v], \mathrm{M}_3[v], \mathrm{M}_4[v])$
13: $\quad$ **else**
14: $\quad\quad \mathsf{E} \leftarrow (f \oplus \mathrm{M}_1[v], \mathrm{M}_2[v], \mathrm{M}_3[v], \mathrm{M}_4[v] \oplus \mathrm{POS}[v-1])$
15: $\quad \mathsf{AST} \leftarrow \mathsf{AST} \cup (\mathsf{pos}, \mathsf{E}, r)$; $v \leftarrow v + 1$
16: **return** $\tau_{\mathsf{add}} = (\mathsf{AST}, \mathsf{key}_1^{(f)}, \mathsf{key}_2^{(f)} \oplus \mathsf{pos})$

Simulation of the deletion token

1: **return** $\tau_{\mathsf{del}} = \mathrm{FK}[f]$

Simulation of the search token

1: $\underline{w} \leftarrow \min(\mathsf{SP}(w))$; $\overline{w} \leftarrow \max(\mathsf{SP}(w))$
2: parse $\mathsf{UpHist}^{>\overline{w}}(\underline{w})$ as $((v_1, \mathbf{add}, f_1), \ldots, (v_T, \mathbf{add}, f_T))$
3: **if** $T = 0 \wedge \mathrm{T_C}[\underline{w}] = \perp$ **then**
4: $\quad$ **return** $\emptyset$
5: **if** $\mathrm{T_C}[\underline{w}] = \perp$ **then**
6: $\quad \mathrm{T_C}[\underline{w}] \leftarrow (\mathbf{0}, \mathbf{0})$
7: **for** each $(v_t, \mathbf{add}, f_t) \in \mathsf{UpHist}^{>\overline{w}}(\underline{w})$ **do**
8: $\quad \mathsf{nhead}^{(\underline{w})} \leftarrow \mathrm{POS}[v_t]$; $\mathsf{nkey}^{(\underline{w})} \xleftarrow{\$} \{0,1\}^\lambda$
9: $\quad r \leftarrow \mathrm{RAND}[v_t]$; $(\mathsf{head}^{(\underline{w})}, \mathsf{key}^{(\underline{w})}) \leftarrow \mathrm{T_C}[\underline{w}]$
10: $\quad$ program H_1 s.t. $\mathbb{H}_1[\mathsf{nkey}^{(\underline{w})}, r] \leftarrow \mathrm{M}_1[v_t]$
11: $\quad$ program H_2 s.t. $\mathbb{H}_2[\mathsf{nkey}^{(\underline{w})}, r] \leftarrow \mathsf{head}^{(\underline{w})} \oplus \mathrm{M}_2[v_t]$
12: $\quad$ program H_3 s.t. $\mathbb{H}_3[\mathsf{nkey}^{(\underline{w})}, r] \leftarrow \mathsf{key}^{(\underline{w})} \oplus \mathrm{M}_3[v_t]$
13: $\quad \mathrm{T_C}[\underline{w}] \leftarrow (\mathsf{nhead}^{(\underline{w})}, \mathsf{nkey}^{(\underline{w})})$
14: $(\mathsf{nhead}^{(\underline{w})}, \mathsf{nkey}^{(\underline{w})}) \leftarrow \mathrm{T_C}[\underline{w}]$
15: **return** $\tau_{\mathsf{srh}} = (\mathsf{nhead}^{(\underline{w})}, \mathsf{nkey}^{(\underline{w})})$

while querying random oracles. The reason for the inconsistency is that the adversary $\mathcal{A}$ may query H_1 with input $(\mathsf{nkey}^{(w)}, r)$ before the entry $\mathbb{H}_1[(\mathsf{nkey}^{(w)}, r)]$ is updated. In this case, it is highly possible that the adversary $\mathcal{A}$ gets response $\mathbf{out}_1 \neq \mathrm{M}_1[w, t_i]$. If such an inconsistency is observed, we call a **Bad** event happens. Note that, the **Bad** event occurs only if the adversary $\mathcal{A}$ can guess correctly the input of H_1. Since both $(\mathsf{nkey}^{(w)}, r)$ are random strings from $\{0,1\}^\lambda$, the probability of a successful guess is $2^{-\lambda}$. Let $\mathsf{poly}(\lambda)$ denote a polynomial function in λ. A PPT adversary can make at most $\mathsf{poly}(\lambda)$ guesses. Hence, we have:

$$\Pr[G_1 = 1] - \Pr[G_2 = 1] \leq \Pr[\mathbf{Bad}] \leq \tfrac{\mathsf{poly}(\lambda)}{2^\lambda}.$$

• **Game** G_3 : This game proceeds similarly as game G_2 to program H_2 and H_3. We replace $e_2 \leftarrow \mathsf{HN}[2] \oplus H_2(\mathsf{nkey}^{(w)}, r)$ and $e_3 \leftarrow \mathsf{HN}[3] \oplus H_3(\mathsf{nkey}^{(w)}, r)$ (line 12 and line 13 in FLY.Addition) with the following codes:

$$1: \ \mathrm{M}_2[w,t] \xleftarrow{\$} \{0,1\}^\lambda; e_2 \leftarrow \mathrm{M}_2[w,t]$$
$$2: \ \mathrm{M}_3[w,t] \xleftarrow{\$} \{0,1\}^\lambda; e_3 \leftarrow \mathrm{M}_3[w,t]$$

where M_2 and M_3 are two maps maintained by the experiment. In the search phase, H_2 and H_3 are programmed in the same way as H_1, except that line 4 in G_2 is replaced by the following codes:

$$1: \ \mathbb{H}_2[\mathsf{nkey}^{(w)}, r] \leftarrow \mathrm{M}_2[w, t_i] \oplus \mathsf{head}_{i-1}^{(w)}$$
$$2: \ \mathbb{H}_3[\mathsf{nkey}^{(w)}, r] \leftarrow \mathrm{M}_3[w, t_i] \oplus \mathsf{key}_{i-1}^{(w)}$$

Using the same argument, we have:

$$\Pr[G_2 = 1] - \Pr[G_3 = 1] \leq 2 \cdot \tfrac{\mathsf{poly}(\lambda)}{2^\lambda}.$$

• **Game** G_4 : This game is slightly different since H_4 is programmed in the updated phase. Specifically, the code $e_4 \leftarrow \mathsf{HN}[4] \oplus H_4(P_{k_3}(f), r)$ (line 14 in FLY.Addition) is replaced with the following codes:

$$1: \ \mathrm{M}_4[w,t] \xleftarrow{\$} \{0,1\}^\lambda; \ \mathbb{H}_4[\mathsf{key}_3^{(f)}, r] \leftarrow \mathrm{M}_4[w,t]$$
$$2: \textbf{if } c = 1 \textbf{ then}$$
$$3: \quad e_4 \leftarrow \mathrm{M}_4[w,t]$$
$$4: \textbf{else}$$
$$5: \quad e_4 \leftarrow \mathrm{M}_4[w,t] \oplus \mathrm{POS}[f, c-1]$$

Using the same argument, we have:

$$\Pr[G_3 = 1] - \Pr[G_4 = 1] \leq \tfrac{\mathsf{poly}(\lambda)}{2^\lambda}.$$

• **Game** G_5 : The DO-side algorithms of G_5 are shown in Alg. 1. The main differences from G_4 lie in the following aspects: (1) This game maintains a map $\mathrm{Updates}$ to record all the update operations since the last search query, and keeps track of the randomly generated strings with a global update counter v. (2) This game utilizes the local state $\mathrm{T_C}$ instead of the map WPK, and postpones the updates of $\mathrm{T_C}$ to the search phase. However, it is hard for the adversary to observe the changes. From the adversary's view, G_4 and G_5 behave exactly the same, since in both games, the outputs of protocols have the same distribution. Hence, we have:

$$\Pr[G_4 = 1] - \Pr[G_5 = 1] = 0.$$

• $\mathbf{Ideal}^{\prod}_{\mathcal{A},\mathcal{S}}(\lambda)$: Given the leakage function collection $\mathcal{L}_{\prod} = (\mathcal{L}^{\mathsf{set}}_{\prod}, \mathcal{L}^{\mathsf{srh}}_{\prod}, \mathcal{L}^{\mathsf{add}}_{\prod}, \mathcal{L}^{\mathsf{del}}_{\prod})$ defined in Theorem 1, we can build a simulator $\mathcal{S}$ as shown in Simulation 1. The difference from G_5 is that in this game, the simulator uses $\underline{w} = \min(\mathsf{SP}(w))$ to denote keyword w. Furthermore, we use notation $\overline{w} = \max(\mathsf{SP}(w))$ to denote the last index of w in $\mathsf{SP}(w)$, and use notation $\mathsf{UpHist}^{>\overline{w}}(\underline{w})$ to denote the partial update history since the last search query. The view

Algorithm 2 Game G_5

FLY+.Setup
1: $(\mathrm{T_C}, \mathrm{T_L}, \mathrm{T_R}, \mathrm{T_P}) \leftarrow$ empty map
2: Updates $\leftarrow$ empty map; Deletes $\leftarrow$ empty set; $v \leftarrow 1$
3: $\sigma \leftarrow \mathrm{T_C}$; $\mathsf{EDB} \leftarrow (\mathrm{T_L}, \mathrm{T_R}, \mathrm{T_P})$
4: **return** (EDB, σ)

FLY+.Addition
1: Append (v, add, f) to Updates$[w]$
2: $\mathsf{pos} \xleftarrow{\$} \{0,1\}^\lambda$; $\tau \xleftarrow{\$} \{0,1\}^\lambda$
3: $\mathrm{POS}[v] \leftarrow \mathsf{pos}$; $\mathrm{TAG}[v] \leftarrow \tau$; $\mathsf{pkey} \leftarrow \mathbf{0}$
4: **for** $j \in [3]$ **do**
5: $\quad \mathrm{M}_j[v] \xleftarrow{\$} \{0,1\}^\lambda$; $e_j \leftarrow \mathrm{M}_j[v]$
6: $\mathsf{E} \leftarrow (e_1, e_2, e_3)$; $v \leftarrow v+1$
7: **return** $\tau_{\mathsf{add}} = (\mathsf{pos}, \mathsf{E}, \tau, \mathsf{pkey})$

FLY+.Deletion
1: Append (v, del, f) to Updates$[w]$
2: find (u, add, f) from Updates$[w]$
3: Deletes $\leftarrow$ Deletes $\cup (u, v)$
4: $\mathrm{TAG}[v] \leftarrow \mathrm{TAG}[u]$; $\mathsf{pos} \leftarrow \mathrm{POS}[u]$
5: $\mathrm{M}_4[v] \xleftarrow{\$} \{0,1\}^\lambda$; $e \leftarrow \mathrm{M}_4[v]$; $v \leftarrow v+1$
6: **return** $\tau_{\mathsf{del}} = (\mathsf{pos}, e)$

FLY+.Search
1: parse Updates$[w]$ as $((v_1, \mathbf{op}_1, f_1), \ldots, (v_T, \mathbf{op}_T, f_T))$
2: **if** $T = 0 \wedge \mathrm{T_C} = \perp$ **then**
3: $\quad$ **return** $\emptyset$
4: **if** $\mathrm{T_C}[w] = \perp$ **then**
5: $\quad \mathsf{lks}^{(w)} \xleftarrow{\$} \{0,1\}^\lambda$; $\mathrm{LKS}[w, 0] \leftarrow \mathsf{lks}^{(w)}$
6: $\quad \mathrm{T_C}[w] \leftarrow (\mathbf{0}, \mathbf{0}, 0, \mathsf{lks}^{(w)})$
7: **for each** $(v_t, \mathbf{op}_t, f_t) \in$ Updates$[w]$ **do**
8: $\quad (\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}^{(w)}) \leftarrow \mathrm{T_C}[w]$
9: $\quad \tau \leftarrow \mathrm{TAG}[v_t]$; $\mathsf{lks}^{(w)} \leftarrow \mathrm{LKS}[w, \mathsf{ac}^{(w)}]$
10: $\quad$ **if** $\mathbf{op}_t = \mathsf{add}$ **then**
11: $\quad\quad \mathsf{nhead}^{(w)} \leftarrow \mathrm{POS}[v_t]$; $\mathsf{nkey}^{(w)} \xleftarrow{\$} \{0,1\}^\lambda$
12: $\quad\quad \mathsf{key}^{(\tau)} \leftarrow \mathsf{SPE.GenKey}(\mathsf{lks}^{(w)}\tau)$
13: $\quad\quad$ **if** $\exists (u, v_t) \in$ Deletes **then**
14: $\quad\quad\quad \mathrm{id} \leftarrow \{0,1\}^\lambda$
15: $\quad\quad$ **else**
16: $\quad\quad\quad \mathrm{id} \leftarrow f_t$
17: $\quad\quad$ program H_1 s.t. $\mathbb{H}_1[\mathsf{nkey}^{(w)}, \mathsf{key}^{(\tau)}] \leftarrow \mathrm{id} \oplus \mathrm{M}_1[v_t]$
18: $\quad\quad$ program H_2 s.t. $\mathbb{H}_2[\mathsf{nkey}^{(w)}, \tau] \leftarrow \mathsf{head}^{(w)} \oplus \mathrm{M}_2[v_t]$
19: $\quad\quad$ program H_3 s.t. $\mathbb{H}_3[\mathsf{nkey}^{(w)}, \tau] \leftarrow \mathsf{key}^{(w)} \oplus \mathrm{M}_3[v_t]$
20: $\quad\quad \mathrm{T_C}[w] \leftarrow (\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}^{(w)})$
21: $\quad$ **else**
22: $\quad\quad (\mathsf{lks}'^{(w)}, \mathsf{pks}^{(\tau)}) \leftarrow \mathsf{SPE.IncPun}(\mathsf{lks}^{(w)}, \tau)$
23: $\quad\quad$ program H_4 s.t. $\mathbb{H}_4[\mathsf{pun}^{(w)}, \tau] \leftarrow \mathsf{pks}^{(\tau)} \oplus \mathrm{M}_4[v_t]$
24: $\quad\quad \mathrm{T_C}[w] \leftarrow (\mathsf{head}^{(w)}, \mathsf{key}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}'^{(w)})$
25: $(\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)}, \mathsf{ac}^{(w)}, \mathsf{lks}^{(w)}) \leftarrow \mathrm{T_C}[w]$
26: **if** $\mathrm{PUN}[w, \mathsf{ac}^{(w)}] = \perp$ **then**
27: $\quad \mathrm{PUN}[w, \mathsf{ac}^{(w)}] \xleftarrow{\$} \{0,1\}^\lambda$
28: $\mathsf{pun}^{(w)} \leftarrow \mathrm{PUN}[w, \mathsf{ac}^{(w)}]$
29: Updates$[w] \leftarrow \perp$; Deletes$[w] \leftarrow \perp$
30: $\mathsf{lks}^{(w)} \xleftarrow{\$} \{0,1\}^\lambda$; $\mathrm{LKS}[w, \mathsf{ac}^{(w)}+1] \leftarrow \mathsf{lks}^{(w)}$
31: $\mathrm{T_C}[w] \leftarrow (\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)}, \mathsf{ac}^{(w)}+1, \mathsf{lks}^{(w)})$
32: **return** $\tau_{\mathsf{srh}} = (\mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)}, \mathsf{lks}^{(w)}, \mathsf{pun}^{(w)})$

produced by $\mathcal{S}$ is indistinguishable from that produced by G_5. Hence, we have:

$$\Pr[\mathbf{Ideal}^{\prod}_{\mathcal{A},\mathcal{S}}(\lambda) = 1] - \Pr[G_5 = 1] = 0.$$

• **Conclusion.** By combing all simulation results, we can say that, for any PPT adversary $\mathcal{A}$, there exists a PRF-adversary $\mathcal{B}$ such that

$$\begin{aligned}&|\Pr[\mathbf{Real}^{\prod}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\prod}_{\mathcal{S},\mathcal{A}}(\lambda) = 1]| \\ &\leq 4 \cdot \mathrm{Adv}^{\mathrm{prf}}_{\mathcal{B}}(\lambda) + 4 \cdot \tfrac{\mathbf{poly}(\lambda)}{2^\lambda}\end{aligned}$$

□

APPENDIX D
PROOF OF THEOREM 2

Proof: The proof is also proceeded by game hops, where hash functions are modeled as random oracles.

• **Game** G_0 : This game is exactly the real-world game except for some bookkeeping changes. The experiment maintains a map CNT to record the number of additions for each keyword w, and a map WIC as the counter of each inserted pair. The following codes are added at the beginning of protocol FLY+.Addition:

1 : **if** $\mathrm{CNT}[w] = \perp$
2 : $\quad \mathrm{CNT}[w] \leftarrow 1$
3 : **end if**
4 : $c \leftarrow \mathrm{CNT}[w]$; $\mathrm{WIC}[w, f] \leftarrow c$; $\mathrm{CNT}[w] \leftarrow c+1$

Furthermore, the experiment utilizes a map WPK to record all the counter/position/key tuples for each keyword w since the last search query. The following codes are added after line 13 of protocol FLY+.Addition:

1 : **if** $\mathrm{WPK}[w] = \perp$ **then**
2 : $\quad$ **Append** $(c-1, \mathsf{head}^{(w)}, \mathsf{key}^{(w)})$ **to** $\mathrm{WPK}[w]$
3 : **Append** $(c, \mathsf{nhead}^{(w)}, \mathsf{nkey}^{(w)})$ **to** $\mathrm{WPK}[w]$

We claim that

$$\Pr[\mathbf{Real}^{\prod}_{\mathcal{A}}(\lambda) = 1] = \Pr[G_0 = 1].$$

• **Game** G_1 : This game replaces the calls to PRFs by picking random strings. The experiment maintains maps, POS, TAG, LKS and PUN. When $Z_{k_{\mathsf{pos}}}(w||f)$ (resp. $F_{k_{\mathsf{pun}}}(w||\mathsf{ac}^{(w)})$, $G_{k_{\mathsf{spe}}}(w||\mathsf{ac}^{(w)})$, or $P_{k_{\mathsf{tag}}}(w||f)$) is required, the experiment checks if $\mathrm{POS}[w, c]$ (resp. $\mathrm{PUN}[w, \mathsf{ac}^{(w)}]$, $\mathrm{LSK}[w, \mathsf{ac}^{(w)}]$, or $\mathrm{TAG}[w, c]$) exists or not. If so, it returns the entry; otherwise, it returns random strings and updates the entry accordingly. If the adversary $\mathcal{A}$ can distinguish between G_0 and G_1, we can build a reduction $\mathcal{B}_1$ to distinguish PRFs from truly random functions. Therefore, we have:

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

• **Game** G_2 : This game differs from G_1 in the following aspects. First, the experiment maintains a map KEY to record the primary key for each pair. The following code

Simulation 2 Simulator $\mathcal{S}$

Simulation of Setup

$(\mathrm{T_C}, \mathrm{T_L}, \mathrm{T_R}, \mathrm{T_P}) \leftarrow$ empty map
$v \leftarrow 1$; $\sigma \leftarrow \mathrm{T_C}$; $\mathsf{EDB} \leftarrow (\mathrm{T_L}, \mathrm{T_R}, \mathrm{T_P})$
return (EDB, σ)

Simulation of the addition token

$\mathsf{pos} \xleftarrow{\$} \{0,1\}^\lambda$; $\tau \xleftarrow{\$} \{0,1\}^\lambda$;
$\mathrm{POS}[v] \leftarrow \mathsf{pos}$; $\mathrm{TAG}[v]) \leftarrow \tau$; $\mathsf{pkey} \leftarrow \mathbf{0}$
for $j \in [3]$ **do**
  $\mathrm{M}_j[v] \xleftarrow{\$} \{0,1\}^\lambda$; $e_j \xleftarrow{\$} \mathrm{M}_j[v]$
$\mathsf{E} \leftarrow (e_1, e_2, e_3)$; $v \leftarrow v + 1$
return $\tau_{\mathsf{add}} = (\mathsf{pos}, \mathsf{E}, \tau, \mathsf{pkey})$

Simulation of the deletion token

find (u, v) from $\mathsf{DelTime}$
$\mathrm{TAG}[v] \leftarrow \mathrm{TAG}[u]$; $\mathsf{pos} \leftarrow \mathrm{POS}[u]$
$\mathrm{M}_4[v] \xleftarrow{\$} \{0,1\}^\lambda$; $e \xleftarrow{\$} \mathrm{M}_4[v]$; $v \leftarrow v + 1$
return $\tau_{\mathsf{del}} = (\mathsf{pos}, e)$

Simulation of the search token

$\underline{w} \leftarrow \min(\mathsf{SP}(w))$; $\overline{w} \leftarrow \max(\mathsf{SP}(w))$
for each $(v_l, f_l) \in \mathsf{TimeDB}(\underline{w})$ **do**
  add $(v_l, \mathbf{add}, f_l)$ into $\mathrm{Updates}[\underline{w}]$
for each $(u_l, v_l) \in \mathsf{DelHist}(\underline{w})$ **do**
  add $(u_l, \mathbf{add}, \perp)$ and $(v_l, \mathbf{del}, \perp)$ into $\mathrm{Updates}[\underline{w}]$
sort $\mathrm{Updates}[\underline{w}]$ in the ascending order of timestamp
parse $\mathsf{UpHist}^{>\overline{w}}(\underline{w})$ as $((v_1, \mathbf{op}, f_1), \ldots, (v_T, \mathbf{op}, f_T))$
if $T = 0 \wedge \mathrm{T_C} = \perp$ **then**
  return $\emptyset$
if $\mathrm{T_C}[\underline{w}] = \perp$ **then**
  $\mathsf{lks}^{(\underline{w})} \xleftarrow{\$} \{0,1\}^\lambda$; $\mathrm{LKS}[\underline{w}, 0] \leftarrow \mathsf{lks}^{(\underline{w})}$
  $\mathrm{T_C}[\underline{w}] \leftarrow (\mathbf{0}, \mathbf{0}, 0, \mathsf{lks}^{(\underline{w})})$
for each $(v_t, \mathbf{op}_t, f_t) \in \mathsf{UpHist}^{>\overline{w}}(\underline{w})$ **do**
  $(\mathsf{head}^{(\underline{w})}, \mathsf{key}^{(\underline{w})}, \mathsf{ac}^{(\underline{w})}, \mathsf{lks}^{(\underline{w})}) \leftarrow \mathrm{T_C}[\underline{w}]$
  $\tau \leftarrow \mathrm{TAG}[v_t]$; $\mathsf{lks}^{(\underline{w})} \leftarrow \mathrm{LKS}[\underline{w}, \mathsf{ac}^{(\underline{w})}]$
  if $\mathbf{op}_t = \mathbf{add}$ **then**
    $\mathsf{nhead}^{(\underline{w})} \leftarrow \mathrm{POS}[v_t]$; $\mathsf{nkey}^{(\underline{w})} \xleftarrow{\$} \{0,1\}^\lambda$
    $(\mathsf{key}^{(\tau)}) \leftarrow \mathsf{SPE.GenKey}(\mathsf{lks}^{(\underline{w})}, \tau)$
    if $f_t = \perp$ **then**
      $\mathrm{id} \leftarrow \{0,1\}^\lambda$
    else
      $\mathrm{id} \leftarrow f_t$
    program H_1 s.t. $\mathbb{H}_1[\mathsf{nkey}^{(\underline{w})}, \mathsf{key}^{(\tau)}] \leftarrow \mathrm{id} \oplus \mathrm{M}_1[v_t]$
    program H_2 s.t. $\mathbb{H}_2[\mathsf{nkey}^{(\underline{w})}, \tau] \leftarrow \mathsf{head}^{(\underline{w})} \oplus \mathrm{M}_2[v_t]$
    program H_3 s.t. $\mathbb{H}_3[\mathsf{nkey}^{(\underline{w})}, \tau] \leftarrow \mathsf{key}^{(\underline{w})} \oplus \mathrm{M}_3[v_t]$
    $\mathrm{T_C}[\underline{w}] \leftarrow (\mathsf{nhead}^{(\underline{w})}, \mathsf{nkey}^{(\underline{w})}, \mathsf{ac}^{(\underline{w})}, \mathsf{lks}^{(\underline{w})})$
  else
    $(\mathsf{lks}'^{(\underline{w})}, \mathsf{pks}^{(\tau)}) \leftarrow \mathsf{SPE.IncPun}(\mathsf{lks}^{(\underline{w})}, \tau)$
    program H_4 s.t. $\mathbb{H}_4[\mathsf{pun}^{(\underline{w})}, \tau] \leftarrow \mathsf{pks}^{(\tau)} \oplus \mathrm{M}_4[v_t]$
    $\mathrm{T_C}[\underline{w}] \leftarrow (\mathsf{head}^{(\underline{w})}, \mathsf{key}^{(\underline{w})}, \mathsf{ac}^{(\underline{w})}, \mathsf{lks}'^{(\underline{w})})$
$(\mathsf{nhead}^{(\underline{w})}, \mathsf{nkey}^{(\underline{w})}, \mathsf{ac}^{(\underline{w})}, \mathsf{lks}^{(\underline{w})}) \leftarrow \mathrm{T_C}[\underline{w}]$
if $\mathrm{PUN}[\underline{w}, \mathsf{ac}^{(\underline{w})}] = \perp$ **then**
  $\mathrm{PUN}[\underline{w}, \mathsf{ac}^{(\underline{w})}] \xleftarrow{\$} \{0,1\}^\lambda$
$\mathsf{pun}^{(\underline{w})} \leftarrow \mathrm{PUN}[\underline{w}, \mathsf{ac}^{(\underline{w})}]$
$\mathsf{lks}^{(\underline{w})} \xleftarrow{\$} \{0,1\}^\lambda$; $\mathrm{LKS}[\underline{w}, \mathsf{ac}^{(\underline{w})} + 1] \leftarrow \mathsf{lks}^{(\underline{w})}$
$\mathrm{T_C}[\underline{w}] \leftarrow (\mathsf{nhead}^{(\underline{w})}, \mathsf{nkey}^{(\underline{w})}, \mathsf{ac}^{(\underline{w})} + 1, \mathsf{lks}^{(\underline{w})})$
return $\tau_{\mathsf{srh}} = (\mathsf{nhead}^{(\underline{w})}, \mathsf{nkey}^{(\underline{w})}, \mathsf{lks}^{(w)}, \mathsf{pun}^{(\underline{w})})$

$\mathrm{KEY}[w, c] \leftarrow \mathsf{key}^{(\tau)}$ is added after line 7 of FLY+.Addition. Second, instead of querying $H_1(\mathsf{nkey}^{(w)}, \mathsf{key}^{(\tau)})$ to obtain a mask in the addition phase (line 9 in protocol FLY+.Addition), the experiment performs as follows:

$$\mathrm{M}_1[w, c] \xleftarrow{\$} \{0,1\}^\lambda; e_1 \leftarrow \mathsf{HN} \oplus \mathrm{M}_1[w, c];$$

where M_1 is a map maintained by the experiment. After line 4 of protocol FLY+.Search, the following codes are added to program the random oracle H_1.

1 : $(c_i, \mathsf{head}_i^{(w)}, \mathsf{key}_i^{(w)})_{i=0}^{T} \leftarrow \mathrm{WPK}[w]$
2 : **for** $i = T$ **to** 1 **do**
3 : $\quad \tau \leftarrow \mathrm{TAG}[w, c_i]$; $\mathsf{key}^{(\tau)} \leftarrow \mathrm{KEY}[w, c_i]$
4 : $\quad \mathsf{nkey}^{(w)} \leftarrow \mathsf{key}_i^{(w)}$
5 : $\quad \mathbb{H}_1[\mathsf{nkey}^{(w)}, \mathsf{key}^{(\tau)}] \leftarrow \mathrm{M}_1[w, c_i]$
6 : **end for**
7 : $\mathrm{WPK}[w] \leftarrow \perp$

G_1 and G_2 behave exactly the same except that in G_2, it is possible for the adversary $\mathcal{A}$ to observe an inconsistency while querying random oracles. If an inconsistency is observed, we call a **Bad** event happens. Note that, the **Bad** event occurs only if the adversary $\mathcal{A}$ can guess correctly the input of H_1. The probability of a successful guess is $2^{-\lambda}$ because both $(\mathsf{nkey}^{(w)}, \mathsf{key}^{(\tau)})$ are random strings from $\{0,1\}^\lambda$. A PPT adversary can make at most $\mathsf{poly}(\lambda)$ guesses, then we have $\Pr[\mathbf{Bad}] \leq \frac{\mathsf{poly}(\lambda)}{2^\lambda}$. Hence, we have:

$$\Pr[G_1 = 1] - \Pr[G_2 = 1] \leq \Pr[\mathbf{Bad}] \leq \frac{\mathsf{poly}(\lambda)}{2^\lambda}.$$

• **Game** G_3 : This game proceeds similarly as game G_2 to program random oracles H_2 and H_3. We replace the codes $e_2 \leftarrow \mathsf{HN}[2] \oplus H_2(\mathsf{nkey}^{(w)}, \tau)$; $e_3 \leftarrow \mathsf{HN}[3] \oplus H_3(\mathsf{nkey}^{(w)}, \tau)$ (lines 10 and 11 in FLY+.Addition) with the following codes:

1 : $\mathrm{M}_2[w, c] \xleftarrow{\$} \{0,1\}^\lambda; e_2 \leftarrow \mathrm{M}_2[w, c]$
2 : $\mathrm{M}_3[w, c] \xleftarrow{\$} \{0,1\}^\lambda; e_3 \leftarrow \mathrm{M}_3[w, c]$

where M_2 and M_3 are two maps maintained by the experiment. In the search phase, H_2 and H_3 are programmed in the same way as H_1, except that line 5 in G_2 is replaced by the following codes:

1 : $\mathbb{H}_2[\mathsf{nkey}^{(w)}, \tau] \leftarrow \mathrm{M}_2[w, c_i] \oplus \mathsf{head}_{i-1}^{(w)}$
3 : $\mathbb{H}_3[\mathsf{nkey}^{(w)}, \tau] \leftarrow \mathrm{M}_2[w, c_i] \oplus \mathsf{key}_{i-1}^{(w)}$

Using the same argument, we have:

$$\Pr[G_2 = 1] - \Pr[G_3 = 1] \leq 2 \cdot \frac{\mathsf{poly}(\lambda)}{2^\lambda}.$$

• **Game** G_4 : This game programs random oracle H_4 in a slightly different way. First, the code $c \leftarrow \mathrm{WIC}[w, f]$ is added at the beginning of the FLY+.Deletion protocol. Then,

we replace the code $e \leftarrow \mathsf{pks}^{(\tau)} \oplus H_4(\mathsf{pun}^{(w)}, \tau)$ at line 3 of FLY+.Deletion with the following codes:

$$\mathrm{M}_4[w, c] \xleftarrow{\$} \{0,1\}^\lambda; e \leftarrow \mathrm{M}_4[w, c]$$

where M_4 is a map maintained by the experiment. We also maintain a map DEL to record the counter/tag/public key share for each deleted pair since the last search query. The following code is added after line 5 of FLY+.Deletion:

$$\textbf{Append } (c, \tau, \mathsf{pks}^{(\tau)}) \textbf{ to } \mathrm{DEL}[w]$$

In the search phase, the following code is added after line 4 of FLY+.Search:

$1:\ (c_i, \tau_i, \mathsf{pks}^{(\tau_i)})_{i=1}^{L} \leftarrow \mathrm{DEL}[w]$
$2:\ \textbf{for } i = L \textbf{ to } 1 \textbf{ do}$
$3:\quad \mathbb{H}_4[\mathsf{pun}^{(w)}, \tau_i] \leftarrow \mathsf{pks}^{(\tau_i)} \oplus \mathrm{M}_4[w, c_i]$
$4:\ \textbf{end for}$
$5:\ \mathrm{DEL}[w] \leftarrow \perp$

Using the same argument, we have:

$$\Pr[G_3 = 1] - \Pr[G_4 = 1] \leq \tfrac{\mathsf{poly}(\lambda)}{2^\lambda}.$$

• **Game** G_5 : The DO-side algorithms of G_5 are shown in Alg. 2. This game differs from G_4 in the following aspects. (1) This game maintains a map $\mathrm{Updates}$ to record all the update operations since the last search query, and adopts a set $\mathrm{Deletes}$ to record the timestamps for each deletion operation and the related addition operation since the last search query. (2) This game keeps track of the randomly generated strings with a global counter v. (3) This game calculates the primary keys and public key shares on the fly in the search phase instead of using maps KEY and DEL. (4) This game utilizes the local state $\mathrm{T_C}$ instead of the map WPK, and postpones the updates of $\mathrm{T_C}$ to the search phase. Besides the above bookkeeping changes, this game replaces the file identifiers of the deleted documents by a random string. This modification is only made for the ciphertexts associated with punctured tags. If the adversary $\mathcal{A}$ can distinguish between G_4 and G_5, we can build a reduction $\mathcal{B}_2$ to break the IND-sPUN-CPA security of SPE by performing at most N encryption queries and d puncture queries. Therefore, we have:

$$\Pr[G_4 = 1] - \Pr[G_5 = 1] \leq d \cdot \mathrm{Adv}^{\mathrm{SPE}}_{\mathcal{B}_2}(\lambda).$$

• $\mathbf{Ideal}^{\prod}_{\mathcal{A},\mathcal{S}}(\lambda)$: Given the leakage function collection $\mathcal{L}_{\prod} = (\mathcal{L}^{\mathsf{set}}_{\prod}, \mathcal{L}^{\mathsf{srh}}_{\prod}, \mathcal{L}^{\mathsf{add}}_{\prod}, \mathcal{L}^{\mathsf{del}}_{\prod})$ defined in Theorem 2, we can build a simulator $\mathcal{S}$ as shown in Simulation 2. The difference from G_5 is that in this game, the simulator uses $\underline{w} = \min \mathsf{SP}(w)$ to denote keyword w. Furthermore, we use notation $\overline{w} = \max(\mathsf{SP}(w))$ to denote the last index of w in $\mathsf{SP}(w)$, and use notation $\mathsf{UpHist}^{>\overline{w}}(\underline{w})$ to denote the partial update history since the last search query. Hence, we have:

$$\Pr[\mathbf{Ideal}^{\prod}_{\mathcal{A},\mathcal{S}}(\lambda) = 1] - \Pr[G_5 = 1] = 0.$$

• **Conclusion.** By combing all simulation results, we can say that, for any PPT adversary $\mathcal{A}$, there exists two adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ such that

$$\begin{aligned}&|\Pr[\mathbf{Real}^{\prod}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\prod}_{\mathcal{S},\mathcal{A}}(\lambda) = 1]| \\ &\leq 4 \cdot \mathrm{Adv}^{\mathrm{prf}}_{\mathcal{B}_1}(\lambda) + 4 \cdot \tfrac{\mathbf{poly}(\lambda)}{2^\lambda} + d \cdot \mathrm{Adv}^{\mathrm{spe}}_{\mathcal{B}_2}(\lambda)\end{aligned}$$

□