SAP-SSE: Protecting Search Patterns and Access Patterns in Searchable Symmetric Encryption

Qiyang Song, Zhuotao Liu, Jiahao Cao, Kun Sun, *Member, IEEE*, Qi Li, *Senior Member, IEEE*, and Cong Wang, *Senior Member, IEEE*

***Abstract*— Searchable symmetric encryption (SSE) enables users to search over encrypted documents in untrusted clouds without leaking the search keywords to the clouds. Existing SSE schemes achieve high search efficiency at the expense of leaking access patterns and search patterns, where clouds can recover a large percentage of queried keywords using the leaked access patterns and search patterns. To prevent clouds from recovering users' keywords, researchers have proposed a number of solutions to protect either search patterns or access patterns. However, none of them can protect both access patterns and search patterns. Moreover, existing SSE schemes cannot work in the generic database setting that allows multiple users to write or read over encrypted documents. In this paper, we propose an efficient searchable symmetric encryption scheme, called SAP-SSE, which protects both access patterns and search patterns in the generic database setting. The main idea of protecting search patterns is to leverage re-encryption cryptosystems to shuffle index entries over multiple clouds. To protect access patterns, we distribute secure indexes to multiple clouds and then propose an index redistribution protocol that allows users to renew index entries in clouds. Furthermore, SAP-SSE provides a configurable security policy to balance security and efficiency. Formal security analysis and experimental evaluation show that SAP-SSE can prevent pattern leakage with low overhead.**

***Index Terms*— Searchable symmetric encryption, access pattern leakage, search pattern leakage.**

Manuscript received March 17, 2020; revised August 3, 2020 and October 17, 2020; accepted November 6, 2020. Date of publication December 2, 2020; date of current version January 5, 2021. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1800304; in part by NSFC under Grant 61572278; in part by the U.S. ONR Grant N00014-16-1-3214 and Grant N00014-18-2893; in part by the U.S. ARO Grant W911NF-17-1-0447; and in part by the Research Grants Council of Hong Kong under Grant CityU 11217819 and Grant CityU 11217620. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Vanesa Daza. *(Corresponding author: Qi Li.)*

Qiyang Song is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, also with the Beijing National Research Center for Information Science and Technology, Beijing 100084, China, and also with the Department of Information Sciences and Technology, George Mason University, Fairfax, VA 22030-4422 USA (e-mail: songqy17@tsinghua.org.cn).

Zhuotao Liu, Jiahao Cao, and Qi Li are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology, Beijing 100084, China (e-mail: zliu48@illinois.edu; caojh15@mails.tsinghua.edu.cn; qli01@tsinghua.edu.cn).

Kun Sun is with the Department of Information Sciences and Technology, George Mason University, Fairfax, VA 22030-4422 USA (e-mail: ksun3@gmu.edu).

Cong Wang is with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: congwang@cityu.edu.hk).

Digital Object Identifier 10.1109/TIFS.2020.3042058

I. INTRODUCTION

CLOUD storage is an outsourcing service for storing explosively growing personal and enterprise data. Recently, the emerging events of data breaches [1] have raised severe privacy concerns for cloud data. Searchable encryption [2] provides a solution to protect private data and search keywords while enabling search functionality. Ideally, searchable encryption can be achieved with a high-security guarantee via oblivious RAM (ORAM) [3]. However, this mechanism has not been widely deployed due to frequent interactions and high communication overhead. To protect data privacy without sacrificing efficiency and searchability, researchers have proposed Searchable Symmetric Encryption (SSE) [4]–[11] to enable clients to perform keyword searches over encrypted documents without leaking search keywords.

The existing SSE schemes (e.g., [4]–[11]) gain efficiency at the expense of some leakage. Typically, SSE leakage consists of access patterns and search patterns. Access patterns reveal the relationship between search operations and matched documents, and search patterns reveal which search operations link to the same keyword. Although those SSE schemes have proven *adaptive security* [4], recent work shows the semi-honest cloud can still recover encrypted search keywords [12]–[15]. By exploiting search patterns [12], the cloud can infer keywords from user search habits. By exploiting access patterns [13]–[15], the cloud can utilize the prior knowledge of documents to derive keywords. Particularly, Zhang *et al.* [15] propose a file-injection attack, achieving 70% keyword recovery accuracy with 20% prior knowledge of documents.

To reduce the risk of keyword disclosures, researchers have proposed several SSE schemes [16]–[20] to minimize the leakage of access patterns and search patterns. None of them can protect access patterns and search patterns simultaneously. However, it is critical to protect the two types of patterns since they are correlated. Namely, when one is leaked, an adversary can derive the information of the other one. For instance, if search patterns are protected, but access patterns are leaked, the cloud can still infer the search patterns of some keywords. This is because a specific keyword may be related to unique documents. When the cloud observes some search operations matching the unique documents, these search operations can be considered to link the same keyword. If access patterns are protected, but search patterns are leaked, the cloud can utilize search habits to recover some keywords. As prior knowledge of documents may show the relationship between specific keywords and matched documents, the cloud can still infer these keywords' access patterns. Therefore, protecting only one pattern type is insufficient to reduce SSE leakage.

Another limitation of existing SSE schemes is that they can only protect access patterns or search patterns in static databases, where only a single user can read over encrypted documents. However, a generic database should allow multiple users to write and read over encrypted documents. The generic database should be further studied as the cloud can infer more sensitive information by observing update operations from specific users [21]. Although we can apply forward-secure SSE schemes [6], [7] to reduce information leakage in update operations, they still reveal update frequency of keywords, which may be used as a stepstone to derive access patterns or search patterns. Therefore, it is critical to design new update operations without leaking patterns. However, supporting the multi-user setting, a prerequisite of generic databases, can not be trivially done by extending existing SSE schemes to protect patterns. Specifically, the multi-user setting [11] requires users to hold distinct keys that could be easily managed, whereas the existing SSE schemes [16], [17] only allow a single user with a key to generate search tokens and hide search patterns. Therefore, a new stack of search operations is required to enable pattern protection for multiple users with distinct keys.

In this paper, we propose an efficient searchable symmetric encryption scheme, called SAP-SSE, which can protect access patterns and search patterns simultaneously in generic databases. Similar to prior work, SAP-SSE adopts encrypted *secure index* [22] to achieve sublinear search complexity. To protect access patterns and search patterns, we design an *index shuffle protocol* and an *index redistribution protocol* that periodically transform the contents, ciphertexts, and locations of index entries in multiple non-colluding clouds. As a result, each cloud will observe a new view of search tokens and search results. Therefore, neither of the clouds can derive search patterns and access patterns.

Within our index shuffle protocol, each cloud leverages re-encryption cryptosystems [23], [24] to re-encrypt index entries with a secret key and then makes a random permutation. Note that search tokens are bound with index entries. Therefore, clouds will observe new search tokens after the process of index shuffle. As the re-encryption cryptosystems do not reveal the plaintexts of index entries and search tokens, each cloud cannot correlate new search tokens to previous ones until it corrupts all other clouds. Therefore, clouds cannot derive search patterns from search tokens.

Our index redistribution protocol's key insight is to divide the original document index between multiple clouds and then rewrite the divided parts after search operations. Afterward, each cloud will observe new search results. To prevent clouds from inferring the relationship between the new results and previous ones by correlating index entries, we also apply the index shuffle protocol to shuffle the rewritten index entries. As a result, clouds cannot derive access patterns from search results.

Based on the above protocols, we design the search and update operations that support pattern protection in generic databases. The critical component of search and update operations is a re-encryption cryptosystem. It allows users to efficiently generate search/update tokens consistent with the currently re-encrypted index entries. Then clouds can transform the tokens and directly use them to query or update indices. As the tokens are not linkable with historical tokens, clouds cannot derive search patterns from these tokens. Moreover, our update operations apply additive homomorphic encryption [25] to merge incremental indices into the original index entry without exposing plaintexts. Therefore, the update frequency of keywords remains unchanged, preventing the possible leakage of update frequency.

Furthermore, we provide an adjustable security policy to balance security and efficiency. The strictest security policy is to execute the index shuffle protocol after each search operation. It is similar to multi-cloud ORAM [26], but with higher efficiency, since it only incurs inter-cloud interactions while multi-cloud ORAM incurs additional frequent client-cloud interactions. Although the inter-cloud interaction overhead is still high, the adjustable security policy allows users to make a trade-off between security and efficiency. In most cases, applications prefer sacrificing non-sensitive information to gain high efficiency, and recent work [12]–[15] shows a small amount of pattern leakage does not breach keyword privacy. Thus, our security policy can satisfy the practical requirements of efficiency and security.

We formulate the security of SAP-SSE and prove the security in the aspects of confidentiality, query unforgeability, and shuffle indistinguishability. Moreover, we conduct experiments to evaluate the security and performance of SAP-SSE. The experimental results demonstrate that SAP-SSE can effectively prevent the keyword recovery attacks with small overhead.

We list the comparison with prior work on several key properties in Table I. To our best knowledge, our scheme is the first SSE scheme that provides optimal protection on patterns while supporting generic databases. Crucially, our scheme achieves these properties with comparable computation and communication complexity (see details in Section IV-H).

In summary, we make the following contributions.

- We propose an efficient SSE scheme called SAP-SSE to protect both access patterns and search patterns in generic databases.
- We design an index shuffle protocol and an index redistribution protocol to periodically transform cloud-side indices. Moreover, we provide multi-user search and update operations supporting pattern protection.
- We provide an adjustable security policy to balance security and performance. This policy allows users to quantitatively customize the parameters in the index shuffle process.
- We conduct a security analysis and experiments to demonstrate the security and performance of SAP-SSE. The results indicate that SAP-SSE can prevent the leakage of access patterns and search patterns with low overhead.

II. BACKGROUND

A. Secure Index

Most SSE schemes utilize a secure index to provide efficient search operations. A secure index is an encrypted inverted

TABLE I
COMPARISON WITH PRIOR WORK ON KEY PROPERTIES

Scheme	Computation		Communication		Protected Patterns		Generic DB
	Search	Update	Search	Update	S.P.	A.P.	Setting
Cash et al. [5]	$O(a_w)$	$O(1)$	$O(n_w)$	$O(1)$	×	×	×
Bao et al. [10]	$O(m \times n)$	$O(1)$	$O(n_w)$	$O(1)$	×	×	✓
Chen et al. [19]	$O(1)$	–	$O(n_w^{f(\epsilon)})_{(f(\epsilon_0)\geq 1)}$	–	×	✓	×
Xu et al. [20]	$O(1)$	–	$O(n_w^{f(\alpha)})_{(f(\alpha)\geq 1)}$	–	×	✓	×
Bösch et al. [17]	$O(m)$	–	$O(n_w)$	–	✓	×	×
Yao et al. [16]	$O(ploy(m))$	–	$O(n_w)$	–	✓	×	×
Ours (SAP-SSE)	$O(1)$	$O(m)$	$O(n_w)$	$O(1)$	✓	✓	✓

The complexities show the efficiency of retrieving documents containing a keyword w or updating documents containing a keyword w. The following notations are used throughout this paper. m (resp. n) is the number of keywords (resp. documents) in the database, and $poly(m)$ is a polynomial of m. n_w is the size of a search result for a keyword w. $f(\epsilon)$ (resp. $f(\alpha)$) is a function depending on a privacy budget ϵ (resp. the size of keyword clusters α). The acronym S.P. (resp. A.P.) stands for 'search patterns' (resp. 'access patterns'). The generic DB setting refers to an SSE scheme that supports multiple users to read or write over encrypted documents.

index that records the relationship between keywords and matched document IDs. It consists of *token field* and *ID field*, where the token field is the pseudo-random string generated from keywords, and the ID field is the ciphertext of document IDs. When users search a keyword, they only need to generate a search token, and then clouds can find and decrypt the ciphertext of matched document IDs with the token.

B. Symmetric Searchable Encryption

Researchers have proposed a variety of SSE schemes [4]–[11] to enable clouds to search over encrypted documents. Particularly, the state of the art can work in the setting of generic databases, which allow multiple users to write and read over encrypted documents. It can be abstracted six algorithms as follows:

- *Setup*(1^k): is run by a system manager. It takes as input a security parameter k and outputs global parameters P.
- *AddUser*(u, P): is run by a system manager. It takes as input a user identity u and a global parameters P, and outputs a secret user key sk_u.
- *UpdToken*(f, sk_u): is run by a user. It takes as input a file f and a user key sk_u, and outputs update tokens α_f.
- *Upd*(α_f, π, D): is run by the cloud. It takes as input update tokens α_f, a secure index π, and a document collection D, and outputs a new index π' and a document collection D'.
- *SrchToken*(w, sk_u): is run by a user. It takes as input a keyword w and a secret user key sk_u, and outputs a search token σ_w for searching.
- *Srch*(σ_w, π, D): is run by the cloud. It takes as input a search token σ_w, a secure index π, and a document collection D, and outputs the matched documents $D(w)$.

C. Pattern Leakage

To clarity what leakage we aim to protect, we formulate access patterns and search patterns. Let Q be a q-query set whose element is in the form of pair (i, w), where i denotes the timestamp of a query, and w denotes a keyword. The leakage can be represented as follows:

- *Access pattern.* It reveals which documents contain a certain keyword. For each queried keyword w, its access pattern is defined as $ap(w) = \{ID(w)\}$, where $ID(w)$ denotes the IDs of documents containing w.
- *Search pattern.* It reveals which queries link to a certain keyword. For each queried keyword w, its search pattern is defined as $sp(w) = \{i|(i, w) \in Q\}$.

D. Proxy Re-Encryption

Proxy re-encryption [23] enables an untrusted proxy to transform a message encrypted by Alice to a new message that can be decrypted by Bob without exposing plaintexts. The cryptosystem is one cornerstone of our index shuffle protocol. It enables clouds to convert the token fields of index entries without revealing keyword plaintexts. A proxy re-encryption cryptosystem can be presented as follows:

- *KeyGen*(1^k): takes as input a security parameter k, and outputs a key pair (pk_u, sk_u).
- *Enc*(m, pk_u): takes as input a plaintext m and a public key pk_u, and outputs a ciphertext C_u.
- *Dec*(C_u, sk_u): takes as input a ciphertext C_u and a secret key sk_u, and outputs a plaintext m.
- *ProxyKeyGen*(sk_{u_1}, sk_{u_2}): takes as input two secret keys sk_{u_1} and sk_{u_2}, and outputs a re-encryption key $pk_{u_1 \to u_2}$.
- *Re-Enc*$(C_{u_1}, pk_{u_1 \to u_2})$: takes as input a ciphertext C_{u_1} encrypted under sk_{u_1} and a re-encryption key $pk_{u_1 \to u_2}$, and outputs a ciphertext C_{u_2} that can be decrypted by sk_{u_2}.

E. Universal Re-Encryption

Universal re-encryption [24] empowers an untrusted proxy to re-randomize a ciphertext to another ciphertext without revealing plaintexts. Unlike proxy re-encryption, universal re-encryption does not change the decryption keys of ciphertexts. It only converts the form of ciphertexts. With universal re-encryption, clouds can re-randomize ID fields without revealing ID plaintexts. A typical universal re-encryption scheme is implemented by additive homomorphic encryption [25]. It can be presented as follows:

- *KeyGen*(1^k): takes as input a security parameter k, and outputs a key pair (pk_u, sk_u).
- *Enc*(m, pk_u, r): takes as input a plaintext m, a public key pk_u, and a random number r, and outputs a ciphertext C_u^r.
- *Dec*(C_u^r, sk_u): takes as input a ciphertext C_u^r and a secret key sk_u, and outputs a plaintext m.
- *Re-Enc*(C_u^r, pk_u, r'): takes as input a ciphertext C_u^r, a public key pk_u, and a random number r', and outputs a new ciphertext $C_u^{r'}$.

III. SAP-SSE Overview

In this section, we present the system overview of SAP-SSE. We articulate the capabilities of adversaries, our design goals, and the system model.

A. Threat Model and Assumptions

In this paper, we consider multiple untrusted clouds that provide searchable encryption. We assume that the clouds are *honest-but-curious* [4], [5]. Namely, they follow the predefined protocols faithfully but have interests to infer sensitive information from the interactions between clouds and users. Particularly, the clouds may attempt to learn sensitive information from encrypted documents. Moreover, they may strive to recover encrypted search keywords according to the leakage of access patterns and search patterns. We assume the clouds are computationally bounded, which means they cannot dedicate infinite computation resources to derive sensitive information.

Meanwhile, we assume that cloud providers do not collude with each other. This assumption is reasonable since different cloud providers are distinct business entities and even direct competitors. Disclosing user data to other entities is a direct violation of many cloud providers' security policies [27]. Moreover, there exist some solutions [28] that enforce the law and economical means to prevent collusions between clouds. In practice, this non-colluding model has been adopted in a wide range of cloud applications, *e.g.,* secure multi-party computation [29] and multi-cloud storage [26].

B. Design Goals

We aim to design an efficient SSE scheme that can simultaneously protect access patterns and search patterns in generic databases. Particularly, it should satisfy the following security and efficiency requirements:

1) *Confidentiality*: Similar to prior work, SAP-SSE prevents clouds from deriving the plaintexts of documents and keywords from encrypted documents and search tokens. Furthermore, SAP-SSE provides an adjustable security policy to thwart the leakage of access and search patterns.
2) *Efficiency*: SAP-SSE achieves sublinear search complexity that is less than $O(m)$, where m is the number of keywords. Moreover, SAP-SSE provides a configurable security policy that allows users to adjust the level of efficiency according to practical security requirements.

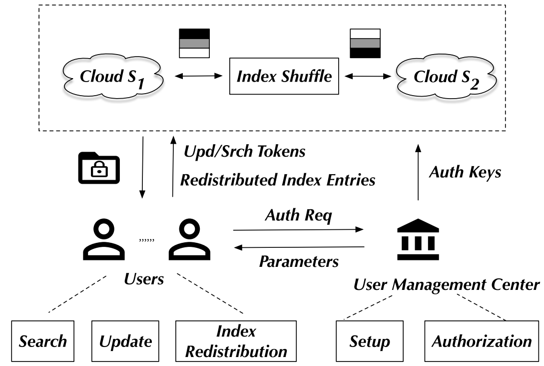


Fig. 1. System architecture.

3) *Query Unforgeability*: SAP-SSE ensures that unauthorized parties cannot forge an honest user's search tokens and update tokens unless they obtain the user's secret key.

C. System Model

Figure 1 shows the architecture of SAP-SSE, which allows multiple users to search and update over encrypted documents. SAP-SSE consists of three parties: (i) a group of *authorized users*, who can execute search and update operations over encrypted documents; (ii) *user management center*, which is in charge of user management; (iii) a set of clouds, which store encrypted documents and also provide search and update operations supporting pattern protection. For simplicity, we use two clouds S_1 and S_2 to present the system; however, our system can be extended to more than two clouds.

The key insight of protecting access and search patterns is to construct secure indices that can be shuffled and redistributed between S_1 and S_2. As Figure 2 shows, secure indices consist of *token field* and *ID field*, which are the ciphertexts of keywords and ID bitmaps. Here, ID bitmaps are n-bit strings that record the IDs of documents containing specific keywords. To protect access patterns, we split the original index into two indices and encrypt them in S_1 and S_2. As the ID fields of the two indices are generated from two split ID bitmaps, users can rewrite ID fields in each cloud to renew the view of search results. Consequently, clouds cannot derive access patterns from search results. In addition, as the token field and ID field are encrypted by re-encryption cryptosystems, each cloud can re-encrypt index entries and permutate them to transform the ciphertexts and locations of index entries. As a result, the corresponding search tokens will be changed, and thus clouds cannot derive search patterns.

Overall, SAP-SSE consists of six main phases: *Setup*, *Authorization*, *Index Shuffle*, *Index Redistribution*, *Search*, and *Update*. We walk through all phases to provide a brief view on the entire workflow.

1) Setup: The user management center chooses a security parameter and then initializes public parameters and master keys. Then, the public parameters are broadcast to clouds, and the master keys are used to authorize users.

2) Authorization: First, a user selects a random number as its secret key for update and search operations. Second,

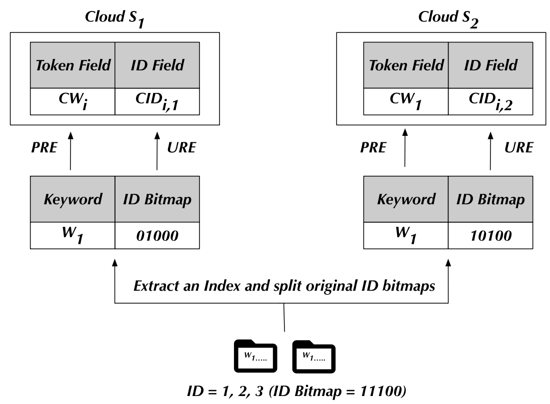


Fig. 2. Index construction. PRE: proxy re-encryption cryptosystem; URE: universal re-encryption cryptosystem.

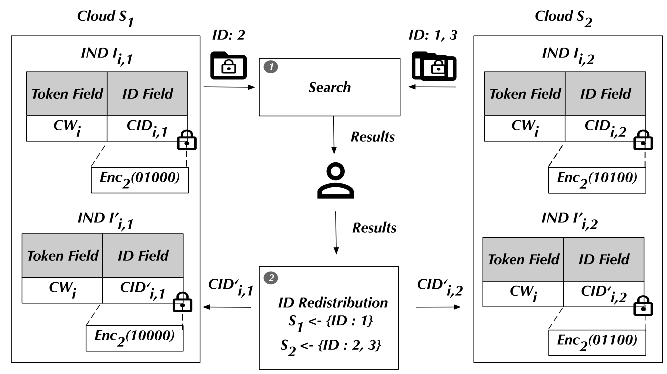


Fig. 4. Index redistribution.

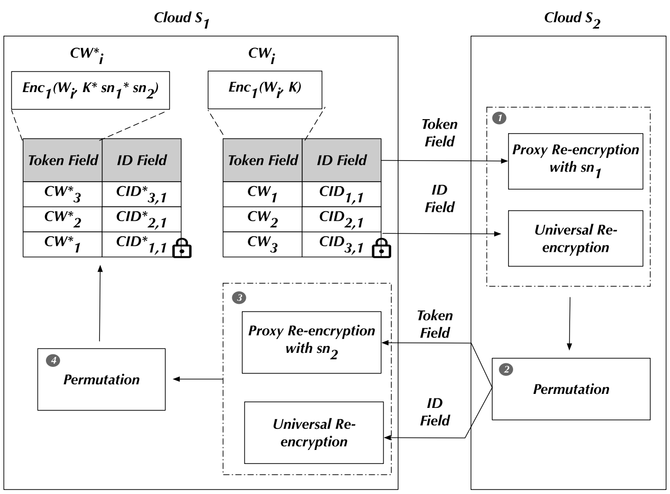


Fig. 3. Index shuffle.

the user management center computes an authorization key according to the user's secret key and then adds the authorization key into clouds. Third, the user management center sends a file key and public parameters to the user.

3) Index Shuffle: Figure 3 shows the index shuffle protocol. As the token field and ID field of index entries are encrypted by a proxy re-encryption scheme and universal re-encryption scheme, respectively. Therefore, each cloud can transform index entries via re-encryption operations and make a random permutation. As clouds use different shuffle numbers sn_1 and sn_2 to re-encrypt index entries, no single cloud can identify the relationship between shuffled index entries and previous ones. As search tokens are bound with token fields, they will be changed when token fields are re-encrypted. As a result, clouds cannot infer search patterns by observing search tokens.

4) Index Redistribution: Figure 4 shows the index redistribution protocol after a search operation. First, a user merges the search results from S_1 and S_2 and then splits the document IDs into two parts. Second, it generates two ID bitmaps and encrypts them according to the two parts. Then, it sends them to replace the ciphertexts of ID fields in S_1 and S_2. Afterward, each cloud will observe new search results. To prevent clouds from correlating the new results from the previous results by observing index entries, we also utilize the index shuffle protocol to transform the locations and ciphertexts of index entries. As a result, clouds cannot derive search patterns.

5) Search: A search operation involves a user and two clouds, and it can be divided into the two stages of search token generation and searching documents in clouds. In the first stage, the user generates the newest search token consistent with currently shuffled index entries, preventing clouds from deriving search patterns. In the second stage, each cloud finds an index entry according to the search token and then extracts an ID field. Next, each cloud obtains a unique set of document IDs via decryption. As each cloud not only stores an encrypted index but also stores encrypted documents, it can directly return a unique set of documents according to observed IDs. Here, we emphasize that each cloud should not decrypt ID fields by itself, since otherwise clouds would learn access patterns of all keywords via decryption. Therefore, we design a new decryption mechanism that can only decrypt ID fields when two clouds are simultaneously involved in search operations.

6) Update: As Figure 2 shows, the original index is decomposed and encrypted in S_1 and S_2. Therefore, update operations should refresh both indices in S_1 and S_2. During an update operation, a user extracts an inverted index from its documents and randomly divides it into two parts. Then, the user generates two sets of update tokens and encrypts documents with a file key. Next, the encrypted documents are sent to all clouds, and the two sets of update tokens are sent to S_1 and S_2, respectively. Finally, each cloud updates its secure index with a set of update tokens and inserts encrypted documents into a document collection. Besides, since the update operations may be exploited by the clouds to infer search patterns when they reveal update frequency or do not possess forward security [15], we design a secure update mechanism to hide the leakage of search patterns (see details in Section IV-G).

IV. SAP-SSE CONSTRUCTION

In this section, we firstly present two re-encryption cryptosystems that are tailored for SAP-SSE construction. We then detail six designed protocols. Next, we compare our construction with existing constructions. Finally, we show how to extend SAP-SSE to support multiple clouds.

A. Tailored Re-Encryption Cryptosystems

In SAP-SSE, our index shuffle protocol is based on a proxy re-encryption cryptosystem [23] and a universal re-encryption cryptosystem [24] to transform the token fields and ID fields of index entries. Since token fields are essentially pseudorandom strings used for searching, we do not need a decryption algorithm. Therefore, to gain high efficiency, we tailor the proxy re-encryption cryptosystem to a lightweight proxy pseudorandom function. As we aforementioned, to prevent the leakage of access patterns, we should not allow ID fields to be arbitrarily decrypted by a cloud. Therefore, we tailor the decryption algorithm of the universal re-encryption cryptosystem.

1) Tailored Proxy Pseudorandom Function (TPF): Our proxy pseudorandom function TPF is based on the one-way encryption algorithm [23]. It is a suite of four algorithms *TPF = (KeyGen, Rnd, RecKeyGen, Rec)*. Let $\mathbb{G}$ be a q-order finite group and $H_{\mathbb{G}}$ be a pseudorandom generation: $\{0, 1\}^* \rightarrow \mathbb{G}$. The proxy pseudorandom function is presented as follows:

- *KeyGen*(1^k): is a probabilistic algorithm to generate a secret key. It takes as input a security parameter k, and outputs a random number $pk \in \mathbb{Z}_q$ as a secret key.
- *Rnd(m, pk)*: is an algorithm to randomize a message. It takes as input a message m and a secret key pk, and computes $ps = H_{\mathbb{G}}(m)^{pk}$. Then, it outputs ps as a pseudorandom string.
- *RecKeyGen*(pk_1, pk_2): is an algorithm to generate a re-encryption key. It takes as input two pseudorandom keys pk_1 and pk_2, and computes $rpk_{1\rightarrow 2} = pk_2/pk_1$. Then, it outputs $rpk_{1\rightarrow 2}$ as a re-encryption key.
- *ReEnc*(ps_1, $rpk_{1\rightarrow 2}$): is an algorithm to re-encrypt a pseudorandom string. It takes as input a pseudorandom string ps_1 and a re-encryption key $rpk_{1\rightarrow 2}$, and computes $ps_2 = ps_1^{rpk_{1\rightarrow 2}}$. Then, it outputs ps_2 as a re-encrypted pseudorandom string.

2) Tailored Universal Re-Encryption (TUR): Our universal re-encryption cryptosystem TUR is based on an additive homomorphic cryptosystem *Paillier = (KeyGen, Enc, Dec, Add)* [25]. To prevent either cloud from decrypting ID fields of index entries by itself, we tailor the decryption algorithm of Paillier to a two-step decryption algorithm. Our universal re-encryption cryptosystem is a suite of six algorithms *TUR = (Setup, KeyGen, Enc, ReEnc, PDec, Dec)*. It works as follows:

- *Setup*(1^k): is a probabilistic algorithm to initialize a master key pair. It takes as input a security parameter k and runs Paillier.KeyGen(1^k) to output a secret key λ and a public key n. Then, it outputs (λ, n) as a master key pair.
- *KeyGen*(λ): is an algorithm to generate two partial decryption keys for two parties. It takes as input a master secret key λ and splits λ to two shares sk_1 and sk_2. Then, it outputs sk_1 and sk_2 as partial decryption keys.
- *Enc(m, n)*: is an algorithm to encrypt a plaintext. It takes as input a plaintext m and a master public key n, and runs Paillier.Enc(m, n) to output a ciphertext c.
- *ReEnc(c, n)*: is an algorithm to re-encrypt a ciphertext. It takes as input a ciphertext c and a master public key n, and computes $c' = c*$ Pailler.Enc(0, n). Then, it outputs c' as a re-encrypted ciphertext.
- *PDec*(c, sk_i): is an algorithm to partially decrypt a ciphertext. It takes as input a ciphertext c and a secret key sk_i, and computes $c_i = c^{sk_i}$. Then, it outputs c_i as a partially decrypted ciphertext.
- *Dec*(c_i, sk_j): is an algorithm to decrypt a ciphertext. It takes a partially decrypted ciphertext c_i and a secret key sk_j, and runs Paillier.Dec(c_i, sk_j) to output a plaintext m.

B. System Setup

In the system setup phase, the user management center needs to initialize global parameters according to a security parameter k. First, it runs the algorithm TPF.KeyGen(1^k) and TUR.Setup(1^k) to generate three master keys K_M, sk_{ID}^M, and pk_{ID}^M, where K_M is the generation key of token fields, sk_{ID}^M is the decryption key of ID fields, and pk_{ID}^M is the encryption key of ID fields. Then, the user management center publishes pk_{ID}^M to all clouds.

Second, the user management center executes the algorithm TUR.KeyGen(sk_{ID}^M) to generate two partial decryption keys sk_{ID}^1 and sk_{ID}^2, and randomly selects two random numbers sn_1 and sn_2 as shuffle numbers. Then, it sends sk_{ID}^1 with sn_1 to cloud S_1 and sends sk_{ID}^2 with sn_2 to cloud S_2, respectively. With the shuffle numbers, the clouds can re-encrypt index entries during the process of index shuffle.

Finally, the user management center selects two random numbers K_f and K_T as a file key and a tag key. The file key is applied to encrypt and decrypt files, and the tag key is used to generate the tags that represent the status of index entries being shuffled.

C. User Authorization

As a user's search tokens and update tokens are generated from a secret key sk_u, they cannot be applied to match the token fields that are encrypted under a master token key K_M. To enable searching and updating, we let clouds re-encrypt search and update tokens to master tokens consistent with token fields. Therefore, user authorization is the process of distributing a re-encryption key into clouds, and authorization keys are essentially re-encryption keys. When the user management center receives a user key sk_u, it runs the algorithm TPF.RecKeyGen(sk_u, K_M) to output an authorization key and then delivers it to clouds. Besides, the user management center sends the file key K_f, the tag key K_T, the shuffle numbers sn_1 and sn_2, and the ID encryption key pk_{ID}^M to the user. With K_f, the user can encrypt and decrypt documents. With the user key, K_T, and pk_{ID}^M, the user can generate update tokens to update documents. With sn_1, sn_2, and a user key, the user can generate search tokens to retrieve documents.

D. Index Shuffle

To protect search patterns, we leverage our proxy pseudorandom function and universal re-encryption to design an

Algorithm 1: Shuffle Index Entries in S_i

Input: our universal re-encryption cryptosystem TUR, our proxy pseudorandom function TPF, a pseudorandom function H_1: $\{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^k$, a set IE_i of index entries stored in S_i, two shuffle numbers sn_1 and sn_2, and a bloom filter BF_i.

Output: a set IE'_i of newly shuffled index entries.

Cloud $\mathbf{S_i}$:
sends to IE_i to S_j;
Cloud $\mathbf{S_j}$:
$TIE \leftarrow \emptyset$;
for *each* $(\gamma, \kappa_i, tag_w^i) \in IE_i$ **do**
  $\gamma' \leftarrow$ TPF.ReEnc$(\gamma,\ sn_j)$;
  $\kappa'_i \leftarrow$ TUR.ReEnc$(\kappa_i,\ pk_{ID}^M)$;
  $tag_w^{i\ \prime} \leftarrow H_1(tag_w^i,\ sn_j)$;
  TIE.insert$(\gamma', \kappa'_i, tag_w^{i\ \prime})$;
randomly permutates TIE, then sends TIE to S_i;
Cloud $\mathbf{S_i}$:
$IE'_i \leftarrow \emptyset$;
for *each* $(\gamma', \kappa'_i, tag_w^{i\ \prime}) \in TIE$ **do**
  $\gamma'' \leftarrow$ TPF.ReEnc$(\gamma',\ sn_i)$;
  $\kappa''_i \leftarrow$ TUR.ReEnc$(\kappa'_i,\ pk_{ID}^M)$;
  $tag_w^{i\ \prime\prime} \leftarrow H_1(tag_w^{i\ \prime},\ sn_i)$;
  BF_i.insert$(tag_w^{i\ \prime\prime})$, IE'_i.insert$(\gamma'', \kappa''_i, tag_w^{i\ \prime\prime})$;
randomly permutates IE'_i;
return IE'_i;

index shuffle protocol. To balance security and efficiency, we also present a quantitative method to evaluate pattern leakage and then provide an adjustable security policy.

1) Two-Cloud Index Shuffle: As the original index is decomposed and encrypted to π_1 and π_2 in two clouds, the entire index shuffle protocol consists of two subroutines that shuffle π_1 and π_2. Here, we present Algorithm 1 to show the subroutine that shuffles π_i. It is collaboratively executed by S_1 and S_2. First, each cloud re-encrypts token fields using proxy re-encryption and re-randomizes ID fields using universal re-encryption. Second, each cloud randomly permutates index entries to hide the locations of index entries. Since each cloud uses secret shuffle numbers to re-encrypt index entries, neither of the clouds can correlate the shuffled index entries to the original ones. As search tokens are bound with index entries, they will be changed after an index shuffle process. Thus, clouds cannot derive search patterns from search tokens.

To help users generate the new search tokens consistent with currently shuffled index entries, we add tags into index entries to reveal shuffle status. In each index shuffle process, S_1 and S_2 re-randomize the tags via a pseudorandom function H_1, and then insert new tags into bloom filters. Therefore, users can simply query bloom filters to learn shuffle status.

2) Leakage Quantification: Intuitively, the access pattern leakage L_{AP} can be indicated by the number of queried keywords. To quantify the search pattern leakage L_{SP}, we need to consider the statistics of search patterns. We notice that clouds can exploit the search frequency of keywords over time, *i.e.,* search frequency vectors, to infer keywords from search patterns [12]. If the search frequency vector of a keyword is more diverged from others, clouds can more accurately recover the keyword. Therefore, we can utilize the divergence of search frequency vectors to quantify L_{SP}. Similar to Liu *et al.* [12], we measure the divergence between two frequency vectors with Euclidean Distance. Therefore, L_{SP} can be quantified as follows:

$$L_{SP} = \frac{1}{|Q|^2} \sum_{i \in Q} \sum_{j \in Q} |fv_i - fv_j|, \tag{1}$$

where Q is a q-query set, and fv_i is the search frequency vector of a keyword over time. In conclusion, pattern leakage can be formulated as a tuple $PL = (L_{AP}, L_{SP})$.

3) Shuffle Policy: The strictest security policy to protect access patterns and search patterns is to shuffle entire index entries after each search operation. However, this security policy is time-consuming. Therefore, we provide an adjusted security policy that allows users to balance security and efficiency. The security policy includes two user-defined leakage thresholds (T_{AP}, T_{SP}), where T_{AP} is the maximum allowable amount of L_{AP}, and T_{SP} is the maximum allowable amount of L_{SP}. If the quantified amount of L_{AP} or L_{SP} reaches the threshold T_{AP} or T_{SP}, then clouds start the index shuffle protocol to shuffle the entire index entries. In practice, the value of L_{AP} and L_{SP} is related to the number of leaked keywords. Therefore, it is possible to conduct experiments on practical datasets to measure this relationship and then define appropriate leakage thresholds in their security policy (see Figure 5.(b) and 5.(c)).

E. Index Redistribution

The index redistribution protocol works after each search operation to re-scatter index entries of a keyword. First, a user merges the search results from two clouds and then randomly re-splits them into two parts. Second, the user records the two parts into ID bitmaps and generates new ID fields according to ID bitmaps. Third, the new fields are sent to clouds to replace the old ones. Afterward, each cloud can observe new search results. To prevent each cloud from correlating the new search results to the old ones, we let clouds cache the new index entries and re-encrypt them in the next index shuffle process.

F. Search Operations

1) Shuffle Status Query: As index entries may be shuffled many times, users should query the shuffle status of index entries to generate search tokens. In each index entry, we add a tag to reveal its shuffle status. During the k-th shuffle process, the tag tag_k in cloud S_i is refreshed to a new tag as follows:

$$tag_{k+1} = H_1(H_1(tag_k, sn_j), sn_i). \tag{2}$$

The tag essentially reveals the numbers of an index entry being shuffled. Since the tags are recorded in the bloom filters BF_1 and BF_2, users can retrieve either one to query an index entry's shuffle status. To query the shuffle status of an index entry, a user needs to generate a set of corresponding tags $\{tag_0,\ tag_1,\ \cdots\}$ with shuffle numbers and test which ones

Algorithm 2: Search Token Generation

Input: a user secret key sk_u, a keyword w, the shuffle status x_w of the index entry related to w, two random shuffle numbers sn_1 and sn_2, and our proxy pseudorandom function TPF.

Output: a search token σ_w.

$t = sk_u * (sn_1 * sn_2)^{x_w}$;
$\sigma_w \leftarrow$ TPF.Rnd(w, t);
return σ_w;

Algorithm 3: Search Documents in S_i

Input: a search token σ_w, the secure index π_j stored in S_j, the document collection C_i stored in S_i, two partial decryption keys sk^1_{ID} and sk^2_{ID} of ID fields, an authorization key $rpk_{u\to M}$, our proxy pseudorandom function TPF, and our universal re-encryption cryptosystem TUR.

Output: a set $C_i(w)$ of encrypted documents.

Cloud $\mathbf{S_j}$:
$\sigma^M_w \leftarrow$ TPF.ReEnc(σ_w, $rpk_{u\to M}$);
$\kappa_j \leftarrow \pi_j$.find($\sigma^M_w$);
$\kappa'_j \leftarrow$ TUR.PDec(κ_j, sk^j_{ID});
sends κ'_j to S_i;
Cloud $\mathbf{S_i}$:
$bm \leftarrow$ TUR.Dec(κ'_j, sk^i_{ID});
finds $C_i(w)$ according to the bitmap bm;
return $C_i(w)$;

Algorithm 4: Update Token Generation

Input: a user secret key sk_u, a file f, a set $\{x_{w_i} \mid w_i \in f\}$ of the index shuffle statuses related to the keywords contained in f, a tag key K_T, a proxy pseudorandom H_1: $\{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^k$, a secret permutation function P_k, two shuffle numbers sn_1 and sn_2, our proxy pseudorandom function TPF, and our universal re-encryption cryptosystem TUR.

Output: two sets α^1_f and α^2_f of update tokens.

$\alpha^1_f \leftarrow \emptyset$, $\alpha^2_f \leftarrow \emptyset$, $bm_1 \leftarrow \{0\}^l$, $bm_2 \leftarrow \{0\}^l$;
for *each* $w \in f$ **do**
    $i \xleftarrow{\$} \{1,2\}$, $bm_i[ID(f)] \leftarrow 1$;
    $\gamma \leftarrow$ TPF.Rnd(w_i, sk_u);
    $\kappa_1 \leftarrow$ TUR.Enc(PK^M_{ID}, bm_1);
    $\kappa_2 \leftarrow$ TUR.Enc(PK^M_{ID}, bm_2);
    tag^1_w, $tag^2_w \leftarrow H_1(K_T, w)$;
    if $x_w > 0$ **then**
        for $i \in \{0, \cdots, x_w - 1\}$ **do**
            $tag^1_w \leftarrow H_1(H_1(tag^1_w, sn_2), sn_1)$;
            $tag^2_w \leftarrow H_1(H_1(tag^2_w, sn_1), sn_2)$;
        $pos_1 \leftarrow P_k(tag^1_w)$, $pos_2 \leftarrow P_k(tag^2_w)$;
        α^1_f.insert(pos_1, κ_1, $\perp$), α^2_f.insert(pos_2, κ_2, $\perp$);
    else
        α^1_f.insert(γ, κ_1, tag^1_w), α^2_f.insert(γ, κ_2, tag^2_w);
return α^f_1, α^f_2;

are contained in BF_i. If tag_k is found in BF_i and tag_{k+1} is not found in BF_i, then we can conclude that the shuffle status is $k+1$, which means the index entry has been shuffled $k+1$ times. Particularly, the status '0' represents that the index entry is newly inserted into secure indices and has not been shuffled. With the shuffle status, the user can generate the search tokens consistent with currently shuffled index entries.

2) Search Token Generation: When index entries are shuffled x_w times, the key of their token fields is transformed from the original value K to the new value $K * (sn_1 * sn_2)^{x_w}$, where sn_1 and sn_2 are shuffle numbers. Therefore, to enable search operations, we need to generate search tokens consistent with currently shuffled index entries. Algorithm 2 shows the detailed operation of search token generation. It generates the search token of a keyword according to a user key sk_u, the shuffle status x_w, and the shuffle numbers sn_1 and sn_2. Essentially, the key of the generated search token is $sk_u * (sn_1 * sn_2)^{x_w}$. By re-encrypting it with the authorization key K/sk_u, clouds can transform the key to $K * (sn_1 * sn_2)^{x_w}$, which is identical to the key of token fields. Therefore, clouds can directly use the search token to query the post-shuffled index entries.

3) Searching Documents: As S_i and S_j store two different index entries, the searching process consists of two different sub-processes in S_i and S_j. To prevent a single cloud from arbitrarily decrypting all ID fields to infer access patterns, we apply the two-step decryption algorithm of our universal re-encryption cryptosystem into each sub-process. As a result, S_i cannot find and decrypt the corresponding document IDs without the assist of S_j. Algorithm 3 shows a detailed searching sub-process in S_j. When receiving a search token, S_j re-encrypts it to a master token with the corresponding authorization key and finds an index entry with the master token. Then, S_j partially decrypts the ID field and sends it to S_i. Next, S_i fully decrypts the ID field and extracts document IDs. Finally, S_i accordingly finds encrypted documents and returns them to a user. The searching sub-process in S_j is similar. In conclusion, the documents retrieved from S_1 and S_2 constitute a complete search result.

G. Update Operations

Note that clouds can recover a large percentage of keywords in non-forward-secure SSE schemes [15]. Therefore, we let S_1 and S_2 jointly update two indices to achieve forward security. An update operation for the index π_1 consists of two stages. S_1 firstly permutates the index π_1 and sends it to S_2, and then S_2 updates the permutated index according to update tokens. As S_1 does not observe update locations, and S_2 does not observe the original index, our update operations achieve forward security. Moreover, we also apply homomorphic encryption to add new IDs into ID fields. Therefore, the update locality of a keyword remains unchanged, preventing the leakage of update frequency. The details are as follows:

1) Update Token Generation: Algorithm 4 shows the process of update token generation when a user uploads a document f. For each keyword $w \in f$, a user initializes two

Algorithm 5: Update a Document in S_i

Input: a set α^i_f of update tokens an encrypted document c, the encrypted document collection C_i stored in S_i, an authorization key $rpk_{u\rightarrow M}$, the secure index π_i stored in S_i, a secret permutation function P_k, a bloom filter BF_i, our proxy pseudorandom function TPF, and our universal re-encryption cryptosystem TUR.

Output: a new secure index π'_i and a new encrypted document collection C'_i.

<u>**Cloud $\mathbf{S_i}$**</u>:
$d \leftarrow \{\}$;
for $(\gamma, \kappa_i, tag^i_w) \in \pi_i$ **do**
  $pos_i \leftarrow P_k(tag^i_w)$; // permutate π_i
  $d[pos_i] \leftarrow$ TUR.ReEnc(κ_i, pk^M_{ID});
sends d to S_j;
<u>**Cloud $\mathbf{S_j}$**</u>:
for $(pos_i, \cdot) \in d$ **do**
  if $(pos_i, \kappa'_i, \cdot) \in \alpha^i_f$ **then**
    $d[pos_i] \leftarrow d[pos_i] \oplus \kappa'_i$; // add $\alpha^i_f \setminus \alpha_{new}$
    α^i_f.del((pos_i, κ'_i, $\cdot$));
  else
    $d[pos_i] \leftarrow$ TUR.ReEnc($d[pos_i]$, pk^M_{ID});
sends d, α^i_f to S_i;
<u>**Cloud $\mathbf{S_i}$**</u>:
sets $\pi'_i \leftarrow \emptyset$;
for $(\gamma, \kappa_i, tag^i_w) \in \pi_i$ **do**
  $pos_i = P_k(tag^i_w)$;
  π'_i.insert(γ, $d[pos_i]$, tag^i_w);
for $(\gamma, \kappa_i, tag^i_w) \in \alpha^i_f$ **do**
  $\gamma \leftarrow$ TPF.ReEnc(γ, $rpk_{u\rightarrow M}$) ; // recover π_i
  π'_i.insert(γ, κ_i, tag^i_w) ; // insert α_{new}
  BF_i.insert(tag^i_w);
$C'_i \leftarrow C_i$.insert(c);
return π'_i, C'_i;

bitmaps and randomly selects one to mark the ID information. Then, it generates two sets of update tokens by the three cryptosystems TPF, TUR, and H_1. Next, it chooses a secret permutation function P_k. To conceal the update locations of index entries, it uses P_k to permutate the original update tokens and also removes their token fields and tag fields. As the update tokens of new keywords ($x_w = 0$) are not related to any location of current index entries, they do not reveal sensitive information. Therefore, the user does not permutate them and remove their ID fields and tag fields.

2) Updating Documents: Algorithm 5 shows the process of updating a document in S_i ($i \in \{1, 2\}$). The process involves two clouds to jointly update the secure index π_i. First, S_i receives a secret permutation function P_k, and S_j receives update tokens α^f_i. Second, S_i permutates its index entries with P_k, removes the original token fields, universally re-encrypts ID fields, and then sends the permutated index to S_j. Third, S_j updates it with the received update tokens via homomorphic addition. Note that the update tokens α_{new} of new keywords do not reveal sensitive information. Therefore, S_j does not update the permutated index with α_{new}. It only updates the permutated index with the update tokens $\alpha^i_f \setminus \alpha_{new}$. Additionally, to prevent S_i from inferring update positions, S_j also needs to transform the remaining index entries that are not updated. Specifically, S_j universally re-encrypts the ID fields of these index entries and sends the entire index back to S_i. Then, S_i recovers the sequence of index entries using P_k and insert new index entries with α_{new}.

H. Comparison with Existing Constructions

Table I summarizes the key property comparison with existing constructions [5], [10], [16], [17], [19], [20]. We can see that SAP-SSE is the only one scheme that can simultaneously protect access patterns and search patterns. Below, we provide a detailed overhead analysis (including both computation and communication) to demonstrate SAP-SSE's efficiency.

1) Computation: As SAP-SSE utilizes the secure index to provide searchable encryption service, it has the sublinear computation complexity of search operations as the most efficient SSE schemes [5], [19], [20]. Note that we provide new update operations that do not change the index locality corresponding to a keyword. Therefore, a search operation only requires a user to generate a search token, and then clouds can find an index entry with only one lookup. Thus, the computation complexity of search operations is optimal, *i.e.,* $O(1)$. Although SAP-SSE applies a more time-consuming cryptographic primitive than a popular large-scale SSE scheme [5], its lower search complexity counterbalances this defect.

During an update operation, a user firstly generates update tokens according to keyword-document pairs, and then clouds will transform whole m index entries and refresh their indices with the update tokens. Therefore, the computation complexity of update operations is $O(m)$. This is not optimal compared to the existing update operations [5], [10]. However, we can update multiple keyword-document pairs through one update operation to improve average complexity in practice. As we demonstrated in our experiments (see Figure 6.(a)), the average time of updating a pair can be drastically reduced using batching updates, and the complexity of updating a single pair is effectively close to $O(1)$.

Unlike prior SSE schemes supporting pattern protection, SAP-SSE does not introduce additional computation overhead in search operations to protect search patterns. In fact, we asynchronously apply the index shuffle protocol to protect search patterns. In other words, the protocol can be executed after a series of search operations. If the protocol is executed when clouds are idle, the search efficiency is not affected. Therefore, although the complexity of index shuffle is $O(m)$, SAP-SSE may incur much less computation than prior schemes in terms of search pattern protection [16], [17]. Additionally, we provide a configurable security policy that allows users to customize the index shuffle protocol and make a trade-off between security and efficiency. We notice that existing SSE schemes do not allow users to adjust the security level of pattern protection. However, non-sensitive information and a small amount of leakage [12]–[15] may not breach user privacy in practice. Therefore, SAP-SSE is more practical than existing SSE schemes to support pattern protection.

2) Communication: Unlike existing schemes, SAP-SSE does not inject bogus files [20] or obfuscate search results [19] to protect access patterns. Therefore, it has the optimal complexity of communication. Specifically, a search operation requires a user to generate a search token and send it to clouds. Therefore, the communication complexity of search operations is $O(1)$. Since a user only needs to upload two update tokens when updating a document containing a keyword, the communication complexity of update operations is $O(1)$. Note that although an index shuffle process incurs $O(m)$ communication between two clouds, the network bandwidth between existing cloud providers is ample and latency is small, because their data centers often peer with many large-scale Internet service providers (thus they are close to Internet core). Therefore, the inter-cloud communication overhead is minor in practice.

I. Extending SAP-SSE to Support Multiple Clouds

SAP-SSE is built in the setting of two clouds, and it can protect search and access patterns via the index shuffle protocol and index redistribution protocol. The security of these protocols relies on the assumption that the clouds do not collude with each other. To relax this security assumption, we can extend the protocols to support k clouds ($k > 2$). The extended index shuffle protocol is similar to the original protocol. It requires k clouds to permutate each cloud's index entries and re-encrypt them with different shuffle numbers. In this way, a single cloud cannot derive search patterns unless it colludes with other $k-1$ clouds. To extend the index redistribution protocol, we decompose and encrypt the original index to k parts and sends them to k clouds, respectively. Then, users can rewrite each cloud's index entries to renew search results. Thus, a single cloud cannot infer access patterns unless it colludes with other $k-1$ clouds. In conclusion, when SAP-SSE is extended from two clouds to k clouds, we can rely on a weaker assumption that each cloud may collude with some clouds but does not collude with all other $k-1$ clouds.

V. Security Analysis

In this section, we provide security proofs to analyze the security of SAP-SSE. First, we define leakage functions that describe leakage information in two forms. Second, we demonstrate the security of SAP-SSE in the aspects of confidentiality, query unforgeability, and shuffle indistinguishability.

A. Leakage Function

SAP-SSE provides a security policy that allows users to adjust the allowable amount of pattern leakage between two shuffle processes. To clarify the pattern leakage, we define a collection of two stateful leakage functions $\mathcal{L} = (\mathcal{L}^{Query}, \mathcal{L}^{Update})$ describing what information leaks. Let $ap_i(w)$ and $sp_i(w)$ be the leaked access patterns and search patterns of a keyword w between the i-th and $(i+1)$-th shuffle process, and let U_w be a boolean bit revealing if the keyword w has been updated. The leakage collection $\mathcal{L}$ can be presented as follows:

- $\mathcal{L}^{Query}(w) = (sp_1(w), ap_1(w), sp_2(w), ap_2(w), \cdots)$
- $\mathcal{L}^{Update}(w) = (U_w)$

B. Confidentiality

The confidentiality of SAP-SSE is captured by the real word versus ideal world formalization [5]. It is parameterized by the leakage collection $\mathcal{L} = (\mathcal{L}^{Query}, \mathcal{L}^{Update})$. More precisely, we define two games $Real_{\mathcal{A}}$ and $Ideal_{\mathcal{A},\mathcal{S}}$ with a simulator $\mathcal{S}$ and an adversary $\mathcal{A}$. The simulator $\mathcal{S}$ can simulate real protocols with the leakage collection. The adversary $\mathcal{A}$ has the view of either one cloud (S_1 or S_2) and can interact with real (or simulated) protocols.

- $Real_{\mathcal{A}}(k)$: $\mathcal{A}$ honestly triggers all operations, *i.e.,* system setup, user authorization, search, update, index shuffle, and index redistribution, and then outputs a bit b.
- $Ideal_{\mathcal{A},\mathcal{S}}(k)$: $\mathcal{A}$ interacts with the simulated protocols generated from $\mathcal{S}$, and then outputs a bit b.

Definition 1 (Confidentiality): We say that SAP-SSE is $\mathcal{L}$-confidential against adaptive chosen keyword attacks (CKA) if for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that

$$|Pr[Real_A(k) = 1] - P[Ideal_{A,S}(k) = 1]| \leq negl(k), \quad (3)$$

where k is a security parameter and $negl(k)$ is a negligible function taking k as a parameter.

Now, we are ready to state the following theorem to demonstrate the confidentiality of SAP-SSE.

Theorem 1: SAP-SSE scheme is $\mathcal{L}$-confidential against adaptive CKA attacks if H_1 and TPF are pseudorandom functions.

Proof 1: We derive some games from $Real_{\mathcal{A}}(k)$ and $Ideal_{\mathcal{A},\mathcal{S}}$ by hopping and construct a simulator $\mathcal{S}$ to simulate the operations of system setup, user authorization, update, search, index shuffle, and index redistribution in each game.

1) Game G_0: We show how $\mathcal{S}$ simulates the operation of system setup. Particularly, the operation is identical to the original operation, except it does not generate a decryption key sk_{ID}^M of ID fields, a generation key K_M of token fields, a file key K_f, and a tag key K_T. Therefore, we have the following equation.

$$Pr[G_0 = 1] = Pr[Real_{\mathcal{A}}(k) = 1]. \quad (4)$$

2) Game G_1: We show how $\mathcal{S}$ simulates the operation of user authorization. When a authorization request is issued from a user u, $\mathcal{S}$ chooses a random number $rpk_{u\to M}^*$ to simulate the corresponding authorization key. Recall that the real-world authorization key is a re-encryption key generated from TPF. Therefore, the advantage of the adversary distinguishing G_1 from G_0 can be reduced to the distinguishing advantage for TPF. More precisely, there exists an adversary $\mathcal{B}_1$ such that

$$Pr[G_1 = 1] - Pr[G_0 = 1] \leq Adv_{TPF,\mathcal{B}_1}^{prf}(k). \quad (5)$$

3) Game G_2: $\mathcal{S}$ simulates the operations of index redistribution and index shuffle similar to the original operations in two clouds, except it simulates TPF and H_1 by selecting two pseudorandom strings in either one cloud. As TPF is simulated as in G_1, the distinguishing advantage between G_2 and G_1 can be reduced to the distinguishing advantange for H_1. Formally, there exists an adversaries $\mathcal{B}_2$ such that

$$Pr[G_2 = 1] - Pr[G_1 = 1] \leq Adv^{prf}_{H_1,\mathcal{B}_2}(k). \quad (6)$$

4) Game G_3: We show how $\mathcal{S}$ simulates update tokens when a document f is being uploaded (the other parts of update operations are identical to the original operations). For each keyword $w \in f$, $\mathcal{S}$ queries if w is updated into the cloud according to $L^{Update}(w)$. If w is new, $\mathcal{S}$ executes the operation of search token generation similar to the original operation, except replacing and simulating TPF and H_1 as in G_2. If w has been updated, $\mathcal{S}$ retrieves a bloom filter from the simulated cloud and queries the bookkeeping of simulated H_1 to get the shuffle status x_w. Then, $\mathcal{S}$ can generate a search token as the origianl operation. As G_3 utilizes TPF and H_1 simulated in G_2, the distinguishing advantange between G_3 and G_2 is zero. Formally, we have the following equation.

$$Pr[G_3 = 1] = Pr[G_2 = 1]. \quad (7)$$

5) Game G_4: We show how $\mathcal{S}$ simulates a search token when a keyword w is being searched (the other parts of update operations are identical to the original operations). Given $L^{Query}(w)$, if $\mathcal{S}$ finds that w has been searched in the period between i-th and $(i+1)$-th shuffle process, $\mathcal{S}$ generates the search token as same as the previous one. Otherwise, $\mathcal{S}$ randomly selects an historical update token $(\gamma^*, \kappa^*, tag^*)$ that is not related to queried keywords. Next, $\mathcal{S}$ queries the shuffle status x_w as in G_3 and then computes $(\gamma^*)^{(sn_1 * sn_2)^{x_w}}$ as a search token. Since G_4 utilizes an update token simulated in G_3, the distinguishing advantange between G_4 and G_3 is zero. Formally, we have the following equation.

$$Pr[G_4 = 1] = Pr[G_3 = 1]. \quad (8)$$

6) Conclusion: All simulation functions in $\mathcal{S}$ can be straightforwardly derived from Game G_4. Therefore, by combining all simulation results, we can say that for any PPT adversary $\mathcal{A}$, there exist two adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ such that

$$|Pr[Real_A(k) = 1] - P[Ideal_{A,S}(k) = 1]| \leq Adv^{prf}_{TPF,\mathcal{B}_1}(k) + Adv^{prf}_{H_1,\mathcal{B}_2}(k). \quad (9)$$

We thus conclude that the right side probability is $negl(k)$ if TPF and H_1 are pseudorandom functions. □

C. Query Unforgeability

We define the query unforgeability by a probabilistic game. In this game, we consider two adversaries: an adversary $\mathcal{A}_S$ who manipulates a cloud and an adversary $\mathcal{A}_U$ who manipulates multiple users. $\mathcal{A}_S$ and $\mathcal{A}_U$ try to forge the honest users' queries (search tokens), and they can query the oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ to obtain a user's practical queries, respectively.

Definition 2 (Query Unforgeability): We say that SAP-SSE achieves query unforgeability if for any user u^*, there exits a PPT adversary $\mathcal{A}_S$ or $\mathcal{A}_U$ such that:

$$Pr[q \in Q_{u^*} \setminus Q'_{u^*} : q \leftarrow \mathcal{A}_S^{O_1}(k) \text{ or } q \leftarrow \mathcal{A}_U^{O_2}(k)] \leq negl(k), \quad (10)$$

where Q_{u^*} denotes all queries from u^*, Q'_{u^*} denotes the queries from $\mathcal{O}_1$ and $\mathcal{O}_2$, k is a security parameter, and $negl(k)$ is a negligible function taking k as a parameter.

Then, we give a security theorem and sketch a proof to demonstrate query unforgeability as follows:

Theorem 2: SAP-SSE scheme achieves query unforgeability if the proxy pseudorandom function TPF is collision-resistant.

Proof 2: First, $\mathcal{A}_U$ or $\mathcal{A}_S$ chooses a target user u^*. Second, we consider that the two adversaries try to forge a query σ = TPF.Rnd(w, sk_u^*) issued from u^*. If $\mathcal{A}_U$ or $\mathcal{A}_S$ can use a forge key sk_A to generate a query σ' = TPF.Rnd(w, sk_A) that is identical to σ, then we can say that $\mathcal{A}_U$ or $\mathcal{A}_S$ succeeds in forging the user's queries. Note that sk_A does not equal to sk_u^*, we can conclude that the probability of the adversary winning this forging game is $negl(k)$ If TPF is collision-resistant. □

D. Shuffle Indistinguishability

We define the shuffle indistinguishability based on ciphertext indistinguishability. Here, we consider a cloud as the adversary $\mathcal{A}$, who tries to learn the relationship between shuffled index entries and original index entries. Let $IE = \{ent_1, ent_2, \cdots\}$ be a set of original index entries and $IE' = \{ent'_1, ent'_2, \cdots\}$ be a set of shuffled index entries, where an index entry ent_i consists of a token field γ_i, an ID field κ_i, and a tag field tag_i.

Definition 3 (Shuffle Indistinguishability): We can say SAP-SSE achieves shuffle indistinguishability if for any $ent_i \in IE$ and $ent'_j \in IE'$, there exists a PPT adversary $\mathcal{A}$ such that:

$$|Pr(\mathcal{A}(ent_i, ent'_j)) - \frac{1}{|IE|}| \leq negl(k), \quad (11)$$

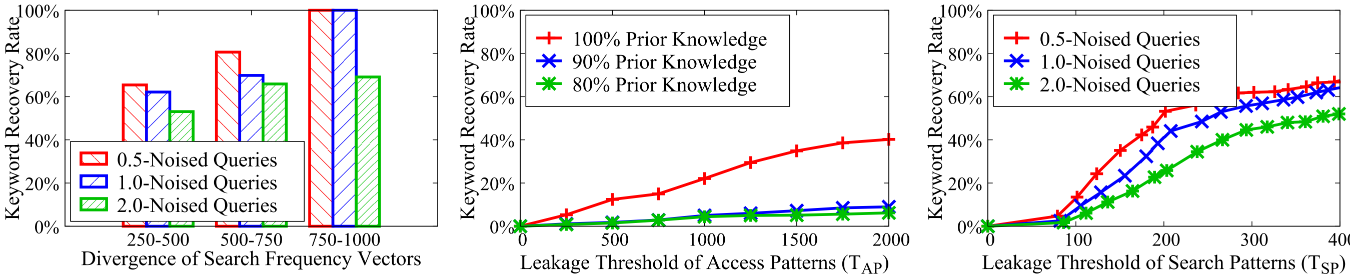
where k is a security parameter and $negl(k)$ is a negligible function taking k as a parameter.

The security of re-encryption cryptosystems and pseudorandom functions underlie the shuffle indistinguishability: if they achieve ciphertext indistinguishability, SAP-SSE achieves shuffle indistinguishability. To demonstrate that, we give a security theorem and sketch a proof as follows:

Theorem 3: SAP-SSE scheme achieves shuffle indistinguishability if TPF is a proxy pseudorandom function, H_1 is a pseudorandom function, and TUR is a CCA-secure universal re-encryption scheme.

Proof 3: Here, we only analyze the index shuffle process for π_1, and the shuffle process for π_2 is similar. Let $Q = (ent_1, ent_2, \cdots)$ be the original index entries, and $Q' = (ent'_1, ent'_2, \cdots)$ be the shuffled index entries. Let ent_i be $(\gamma_i, \kappa_i, tag_i)$. After a shuffle period, ent_i is shuffled to $ent'_i = (\gamma'_i, \kappa'_i, tag'_i)$, where:

- γ'_i = TPF.ReEnc(TPF.ReEnc$(\gamma_i, sn_2), sn_1)$,



(a) Validity of Quantitative Method (b) Thwarting Access-pattern Attacks (c) Thwarting Search-pattern Attacks

Fig. 5. SAP-SSE Security.

- $\kappa_i' = \text{TUR.ReEnc}(\text{TUR.ReEnc}(\kappa_i,\ pk_{ID}^{M}),\ pk_{ID}^{M})$,
- $tag_i' = H_1(H_1(tag_1, sn_2), sn_1)$.

If TPF is a proxy pseudorandom function and H_1 is a pseudorandom function, S_i or S_j cannot distinguish γ_i' and tag_i' of a shuffled index entry from γ_j' and tag_j' of another shuffled index entry without knowing the other cloud's shuffle number. Since clouds choose different random numbers to re-randomize ID fields, if TUR is a CCA-secure universal re-encryption scheme, each cloud cannot distinguish κ_i' from κ_j' without knowing the other cloud's decryption key and random numbers. Therefore, S_i and S_j cannot recognize which shuffled index entry is related to an original index entry. In conclusion, the probability of the adversary $\mathcal{A}$ identifying that the shuffled index entry ent_j' links to the original index entry ent_i is $1/|IE|$. □

VI. System Evaluation

We implement a prototype of SAP-SSE and conduct experiments to evaluate its performance and security strength. The experiments are performed on a PC running Ubuntu 16.04 with four Intel Core i5 2.3GHz processors and 8GB RAM. To measure the feasibility of our scheme in practice, we use real-world Enron email dataset .[1] We randomly select 4742 documents from this dataset as our corpus, which contains 1 million keyword-document pairs and 79101 keywords. To facilitate the search functionality, we utilize Porter Stemmer [30] to extract keyword stems from documents and filter all meaningless stopwords,[2] such as "the", "of" in our keyword space.

Moreover, we simulate user search data to prove the security strength of SAP-SSE. We crawl user search data of 3000 keywords between January 01, 2018, and January 07, 2018, from Google Trend[3] and simulate user search habits. We notice that in a practical database, user queries do not always perfectly match their search habits. Therefore, we follow the prior work [12], [13] to simulate a query dataset by adding Gaussian noise $N(0, \beta * \sigma^2)$ to the search habits, where σ is the standard deviation of the search frequency in different periods, β is the noise coefficient. We simulate three query datasets with a 0.5 noise coefficient, 1.0 noise coefficient, and 2.0 noise coefficient. Then, we use the three datasets to evaluate the security and performance of our scheme.

[1]Enron email dataset. https://www.cs.cmu.edu/ ./enron/

[2]Google stopwords. https://code.google.com/archive/p/stop-words/

[3]Google trends. https://trends.google.com/trends/

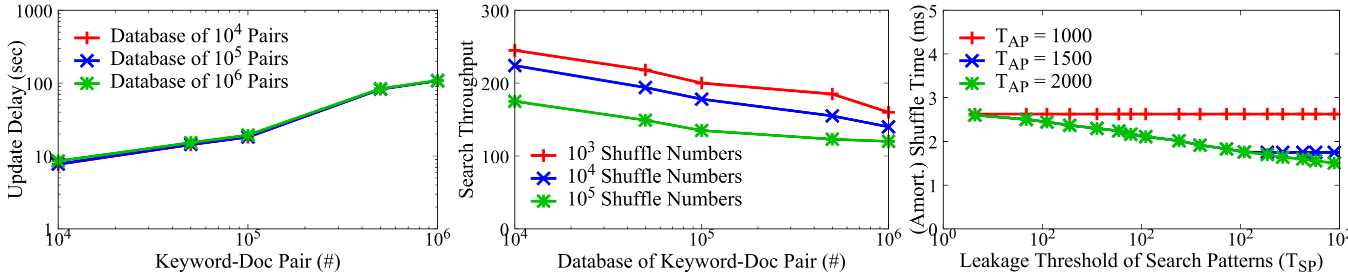
A. Thwarting Keyword Recovery Attacks

Our quantitative method for pattern leakage assumes the divergence of search frequency vectors can reflect the leakage of search patterns: if the search frequency vector of a keyword is more diverged from others, the keyword can be more easily recovered. To validate our quantitative method, we exploit the attack in [12] to recover the keywords of three simulated query datasets. Figure 5.(a) shows the validity of our quantitative method. We can observe that the keyword recovery rate increases along with the divergence of search frequency vectors. When the divergence is between 750 and 1000, the keyword recovery rates on 0.5-noised and 1.0-noised query datasets approximate 100%. Therefore, our quantitative is valid, which means the divergence can indeed be used to evaluate the amount of search pattern leakage.

Now we show the impact of a user-defined security policy. The security policy defines the threshold T_{AP} of access pattern leakage and the threshold T_{SP} of search pattern leakage, respectively. Thus, we separately study how the two thresholds thwart keyword recovery attacks. We duplicate an access-pattern attack [14] and a search-pattern attack [12], and measure their keyword recovery rates under different leakage thresholds. In the access-pattern attack, clouds exploit access patterns with 100%, 90%, and 80% prior knowledge of documents to recover keywords. In the search-pattern attack, clouds exploit search patterns with user search habits to recover keywords. Figure 5.(b) and 5.(c) show that the keyword recovery rate increases along with T_{AP} and T_{SP}. Besides, we can observe low thresholds can effectively thwart keyword recovery attacks. Particularly, T_{AP} of 500 degrades the recovery rate to the range between 0.5% and 4%, and T_{SP} of 80 degrades the recovery rate to the range between 0.5% and 4%. In the following, we show the two thresholds only incur 2.5ms shuffle overhead.

B. Performance of SAP-SSE

1) Update Delay: Our scheme achieves the forward security and hides update frequency by homomorphically adding update tokens to index entries and universally re-encrypting ID fields. To reduce the update delay, we let clouds generate universal re-encryption ciphertexts ahead of time and then transform ID fields of index entries during update operations. Figure 6.(a) shows the delay of updating different keyword-document pairs on three database scales. We can observe that the update delay of the database containing



(a) Update Delay (b) Search Throughput (c) Amortized Shuffle Time Per Search

Fig. 6. SAP-SSE Performance.

10^6 keyword-document pairs is only 1-2% longer than that of other databases. Particularly, updating 10^6 keyword-document pairs (4742 documents) in three databases only incurs the overhead of about 106 seconds, which is an acceptable update overhead in practice.

2) Search Throughput: Figure 6.(b) shows the search throughput in three different database scales. Since SAP-SSE achieves sublinear search complexity, the search throughput does not degrade significantly along with the database size increases. We notice that the search throughput also varies with the shuffle frequency. Namely, if an index entry is shuffled more times, users need more time to query its shuffle status before the search token generation. As the hash function applied for querying shuffle statuses is much faster than the search token generation, we can observe that a number of index shuffle processes only incur slightly higher search overhead. In the worst case, when searching keywords in the 10^6-pair database that has been shuffled 10^5 times, users can still perform 120 search operations within a second. This results prove SAP-SSE is practical for middle-scale databases.

3) Amortized Shuffle Time: Figure 6.(c) shows the amortized shuffle overhead under different security policies. Overall, our security policy determines the amortized shuffle time. If we set the higher leakage thresholds T_{AP} and T_{SP}, then clouds consume lower amortized shuffle overhead. Particularly, combining Figure 6.(c) with Figure 5.(b) and 5.(c), we can see that the security policy can only incur 2.5ms amortized shuffle overhead when the security policy degrades the keyword recovery rate to the range between 0.3% and 4%.

Moreover, we note that only one threshold can determine the amortized shuffle overhead at a point. Namely, if the number of queries limited by T_{SP} is more than the number of queries limited by T_{AP}, then T_{AP} is the key factor in determining the amortized shuffle overhead. For instance, in the case that $T_{AP} = 1500$, the amortized shuffle overhead decreases along with T_{SP} when $T_{SP} < 360$, but it is fixed to 0.0012s when $T_{SP} \geq 360$. Therefore, users can define the optimal $T_{SP} = 360$ when $T_{AP} = 1500$. Similarly, when $T_{AP} = 1000$, T_{SP} ranging from 30 to 420 does not affect the amortized search time. Thus, the optimal value of T_{SP} is 30.

C. Comparison with Existing SSE Implementations

We conduct experiments to compare our implementation with existing SSE implementations [5], [10], [17], [19], [20]. As Yao *et al.* [16] apply infeasible indistinguishability obfuscation to protect search patterns, we do not consider this scheme in our experiments. In our experiments, all search and update operations are performed over the database of 10^6 keyword-document pairs. Each update operation uploads a document containing 10^4 keyword-document pairs, and we measure the average time of updating a pair.

As the schemes in [19] and [20] apply differential privacy and padding approaches to protect access patterns, they may not wholly protect access patterns unless incurring communication overhead as same as the size of entire documents. To reveal the communication overhead in practice, we set their parameters to defend 95% keyword recovery attacks, which can satisfy the security requirement in most cases.

TABLE II

COMPARISON WITH EXISTING SSE SCHEMES THAT SUPPORT PATTERN PROTECTION

		[19]	[20]	[17]	**Ours**
Comp. (ms)	Search	0.08	0.09	92.01	2.80
	Update	–	–	–	1.21
Comm. (KB)	Search	311.30	361.20	56.50	53.00
	Update	–	–	–	23.40
Protect A.P.		✓	✓	×	✓
Protect S.P.		×	×	✓	✓
Generic DB Setting		×	×	×	✓

Table II shows a comparison of our scheme with existing SSE schemes that support pattern protection. We can see that our scheme is the only one that can simultaneously protect search and access patterns, and it can be applied to generic databases. Moreover, SAP-SSE provide pattern protection while maintaining high efficiency. Compared to the scheme [17] that can only protect search patterns, our schemes provide much faster search operations and support additional update operations. Although the schemes [19] and [20] provide faster search operations that can protect access patterns, the built-in differential privacy and padding approaches may not wholly hide access patterns. Additionally, they do not support update operations and incur much higher communication overhead.

Table III shows a comparison of our scheme with some typical SSE schemes that do not support pattern protection. We can observe that our scheme achieves much higher search efficiency and slightly higher update efficiency than the typical scheme [10] that supports generic databases. Additionally, compared to the classic scheme [5] that does not support generic databases, our scheme achieves comparable search efficiency and update efficiency.

TABLE III
COMPARISON WITH TYPICAL SSE SCHEMES WITHOUT ENFORCING PATTERN PROTECTION

		[5]	[10]	**Ours**
Comp. (ms)	Search	0.15	101.12	2.80
	Update	0.11	2.21	1.21
Comm. (KB)	Search	53.50	51.50	53.00
	Update	10.20	64.70	23.40
Generic DB Setting		×	✓	✓

VII. RELATED WORK

A. Searhable Symmetric Encryption

Curtmola *et al.* [4] propose the first SSE scheme and provide formal security definitions. The scheme allows a user to outsource its documents for sharing, and other users can search over encrypted documents with sublinear search complexity. However, the scheme does not provide update operations and supports multiple users. Following this work, a series of SSE schemes have been proposed to provide more functionalities. For example, dynamic SSE schemes [5]–[7] support efficient update operations over the encrypted documents. Multi-user SSE schemes support multiple users [4], [10], [11] to perform search operations and update operations. Recently, researchers [8], [9] have proposed some SSE schemes supporting rich queries. Besides, researchers also [31]–[33] attempt to enhance SSE security. For instance, they have applied ORAM techniques to protect access patterns, search patterns, and communication volume. However, ORAM techniques incur high communication overhead, a number of interaction rounds, and large client storage. Especially when storing large documents, ORAM techniques result in severe performance degradation [34].

B. SSE Leakage

As ORAM techniques are not infeasible in practice, researchers attempt to sacrifice the protection of access and search patterns to achieve efficient SSE schemes. The leakage of access and search patterns is often regarded as a reasonable leakage. Compared to static schemes, existing dynamic SSE schemes leaks more sensitive information. For instance, the non-forward-secure schemes [5] can reveal the relation between index entries and previous search tokens through an update operation, and the forward-secure schemes [6], [7] can leak the update frequency of keywords during search operations. Note that both update frequency and non-forward privacy can be exploited to derive search patterns. Therefore, the leakage of dynamic schemes needs to be further studied. Moreover, since some schemes expose specific plaintext properties to support rich queries, they often leak additional information to clouds, such as plaintext orders [35]. In this paper, we focus on providing protection on access patterns and search patterns since they are two typical types of leakage information, and their leakage is often interleaved.

C. SSE Attacks

Researchers have proposed a series of attacks to infer encrypted keywords in SSE schemes. The most popular attack approaches are exploiting access patterns or search patterns with prior knowledge to recover keywords. For instance, Isam *et al.* [13] propose the first access-pattern attack with prior knowledge of entire documents. Then, Cash *et al.* [5] present an improved attack, achieving the same recovery rate with less prior knowledge. Recently, Zhang *et al.* [15] exploit the leakage of update operations and inject some files to clouds, which achieves a much higher keyword recovery rate in non-forward-secure SSE schemes. Liu *et al.* [12] propose the first search-pattern attack that derives sensitive information from search patterns. Additionally, there are some emerging attacks [35] that exploit other leakage information in SSE schemes, *e.g.,* plaintext orders and communication volume.

D. Access and Search Pattern Protection

Recently, researchers have proposed new SSE schemes to reduce the leakage of access and search patterns. Existing SSE schemes protect search patterns by applying heavy cryptographic tools, which incur prohibitively high communication overhead. For instance, Bosch *et al.* [17] propose a distributed SSE scheme that can protect search patterns over multiple clouds. However, the whole index should be obfuscated by a proxy server before each search operation. That results in a high search complexity. Liu *et al.* [12] utilize the indistinguishability obfuscation (IO) and chameleon hash to protect search patterns. However, inefficient IO functions limit search throughput. Although Li *et al.* [18] claim that their MPC-based SSE scheme can efficiently protect search patterns, the internal clouds can still observe search patterns. In addition, there are several SSE schemes that can protect access patterns, but neither of them can completely protect access patterns. Particularly, they apply differential privacy [19] and padding approaches [20] to make a trade-off between security and efficiency. Thus, clouds can still derive some sensitive information from access patterns.

VIII. CONCLUSION

In this paper, we design an efficient symmetric searchable symmetric encryption, called SAP-SSE, which protects both access patterns and search patterns in generic databases. We present an index shuffle protocol and an index redistribution protocol that can periodically change the contents, locations, and ciphertexts of index entries across multiple clouds. Then, we provide multi-user update operations and search operations that support pattern protection. Furthermore, we design a quantitative method to evaluate pattern leakage and allows users to define a security policy to balance security and efficiency. We conduct a security analysis and experiments to evaluate the security and performance of SAP-SSE. The security proofs and experimental results show that SAP-SSE can effectively thwart attacks with acceptable overhead.

REFERENCES

[1] T.-S. Chou, "Security threats on cloud computing vulnerabilities," *Int. J. Comput. Sci. Inf. Technol.*, vol. 5, no. 3, p. 79, 2013.

[2] D. Xiaoding Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE Symp. Secur. Privacy. (S P)*, May 2000, pp. 44–55.

[3] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, May 1996.
[4] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proc. CCS*, 2006, pp. 79–88.
[5] D. Cash *et al.*, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.
[6] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1465–1482.
[7] 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. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1449–1463.
[8] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *Proc. ESORICS*. Cham, Switzerland: Springer, 2015, pp. 123–145.
[9] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system," in *Proc. VLDB*, 2019, pp. 1664–1678.
[10] F. Bao, R. H. Deng, X. Ding, and Y. Yang, "Private query on encrypted data in multi-user settings," in *Proc. ISPEC*. Heidelberg, Germany: Springer, 2008, pp. 71–85.
[11] M. R. Asghar, G. Russello, B. Crispo, and M. Ion, "Supporting complex queries and access policies for multi-user encrypted databases," in *Proc. ACM workshop Cloud Comput. Secur. Workshop (CCSW)*, 2013, pp. 77–88.
[12] C. Liu, L. Zhu, M. Wang, and Y.-A. Tan, "Search pattern leakage in searchable encryption: Attacks and new construction," *Inf. Sci.*, vol. 265, pp. 176–188, May 2014.
[13] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proc. NDSS*. Reston, VI, USA: Internet Society, 2012, p. 12.
[14] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 668–679.
[15] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *Proc. USENIX Secur.* Berkeley, CA, USA: USENIX Association, 2016, pp. 707–720.
[16] J. Yao, Y. Zheng, C. Wang, and X. Gui, "Enabling search over encrypted cloud data with concealed search pattern," *IEEE Access*, vol. 6, pp. 11112–11122, 2018.
[17] C. Bosch *et al.*, "Distributed searchable symmetric encryption," in *Proc. 12th Annu. Int. Conf. Privacy, Secur. Trust*, Jul. 2014, pp. 330–337.
[18] J. Li, D. Lin, A. C. Squicciarini, J. Li, and C. Jia, "Towards privacy-preserving storage and retrieval in multiple clouds," *IEEE Trans. Cloud Comput.*, vol. 5, no. 3, pp. 499–509, Jul. 2017.
[19] G. Chen, T.-H. Lai, M. K. Reiter, and Y. Zhang, "Differentially private access patterns for searchable symmetric encryption," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 810–818.
[20] L. Xu, X. Yuan, C. Wang, Q. Wang, and C. Xu, "Hardening database padding for searchable encryption," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2019, pp. 2503–2511.
[21] C. V. Rompay, R. Molva, and M. Önen, "A leakage-abuse attack against multi-user searchable encryption," *PoPETs*, vol. 2017, no. 3, p. 168, 2017.
[22] E.-J. Goh, "Secure indexes," *Cryptol. ePrint Arch.*, vol. 2003, p. 216, Oct. 2003.
[23] M. Blaze, G. Bleumer, and M. Strauss, "Divertible protocols and atomic proxy cryptography," in *Proc. EUROCRYPT*. Heidelberg, Germany: Springer, 1998, pp. 127–144.
[24] P. Golle, M. Jakobsson, A. Juels, and P. F. Syverson, "Universal re-encryption for mixnets," in *Proc. CT-RSA*. Heidelberg, Germany: Springer, 2004, pp. 163–178.
[25] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. EUROCRYPT*. Heidelberg, Germany: Springer, 1999, pp. 223–238.
[26] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen, "S 3 ORAM: A computation-efficient and constant client bandwidth blowup ORAM with Shamir secret sharing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 491–505.
[27] G. Cloud. (2020). *Privacy*. [Online]. Available: https://cloud.google.com/security/privacy
[28] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel, "Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 211–227.
[29] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: A system for secure multi-party computation," in *Proc. 15th ACM Conf. Comput. Commun. Secur. (CCS)*, 2008, pp. 257–266.
[30] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 40, no. 3, pp. 211–218, Jul. 2006.
[31] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption," in *Proc. CRYPTO*. Heidelberg, Germany: Springer, 2016, pp. 563–592.
[32] S. Kamara, T. Moataz, and O. Ohrimenko, "Structured encryption and leakage suppression," in *CRYPTO*. Cham, Switzerland: Springer, 2018, pp. 339–370.
[33] S. Kamara and T. Moataz, "Computationally volume-hiding structured encryption," in *Proc. EUROCRYPT*. Cham, Switzerland: Springer, 2019, pp. 183–213.
[34] M. Naveed, "The fallacy of composition of oblivious RAM and searchable encryption," in *Proc. Cryptol. ePrint Arch. (IACR)*, 2015, p. 668.
[35] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-abuse attacks against order-revealing encryption," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 655–672.

Qiyang Song received the master's degree from the Department of Computer Science and Technology, Tsinghua University. He has ever been a Visiting Scholar at George Mason University. His current research interests include cloud privacy and machine learning security.

Zhuotao Liu received the B.S. degree from Shanghai Jiao Tong University and the Ph.D. degree from the University of Illinois at Urbana-Champaign. He is currently a tenure-track Assistant Professor at Tsinghua University. Prior to Tsinghua, he was a TechLead at Google, managing massive-scale software-defined datacenter networks. His research interests are network security & privacy, blockchain infrastructure, datacenter networking, and systems security.

Jiahao Cao received the B.Eng. degree from the Beijing University of Posts and Telecommunications in 2015, and the Ph.D. degree from Tsinghua University in 2020. His current research interests include SDN security, container security, and network traffic analysis.

Kun Sun (Member, IEEE) received the Ph.D. degree from the Department of Computer Science, North Carolina State University. He is currently an Associate Professor with the Department of Information Sciences and Technology, George Mason University. He is also the Associate Director of the Center for Secure Information Systems and the Director of the Sun Security Laboratory, George Mason University. He has more than 15 years of working experience in both industry and academia on systems and network security.

Qi Li (Senior Member, IEEE) received the Ph.D. degree from Tsinghua University. He has ever worked with ETH Zurich and the University of Texas at San Antonio. He is currently an Associate Professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include network and system security, particularly in Internet and cloud security, mobile security, and big data security. He is currently an editorial board member of the IEEE TDSC and ACM DTRAP.

Cong Wang (Senior Member, IEEE) is currently an Associate Professor with the Department of Computer Science, City University of Hong Kong. His current research interests include data and network security, blockchain and decentralized applications, and privacy-enhancing technologies. He is one of the Founding Members of the Young Academy of Sciences of Hong Kong. He received the Outstanding Researcher Award (junior faculty) in 2019, the Outstanding Supervisor Award in 2017, and the President's Awards in 2019 and 2016, respectively, all from the City University of Hong Kong. He is a co-recipient of the IEEE INFOCOM Test of Time Paper Award 2020, the Best Student Paper Award of the IEEE ICDCS 2017, and the Best Paper Award of the IEEE ICPADS 2018 and MSN 2015. He serves/has served as an Associate Editor for the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, the IEEE INTERNET OF THINGS JOURNAL, and the IEEE NETWORKING LETTERS, and the *Journal of Blockchain Research*, and the TPC Co-Chair for a number of IEEE conferences/workshops.