Searchable Symmetric Encryption with Forward Search Privacy

Jin Li, Yanyu Huang, Yu Wei, Siyi Lv, Zheli Liu*, Changyu Dong and Wenjing Lou, *Fellow, IEEE*

Abstract—Searchable symmetric encryption (SSE) has been widely applied in the encrypted database for queries in practice. Although SSE is powerful and feature-rich, it is always plagued by information leaks. Some recent attacks point out that forward privacy which disallows leakage from *update* operations, now becomes a basic requirement for any newly designed SSE schemes. However, the subsequent *search* operations can still leak a significant amount of information. To further strengthen security, we extend the definition of forward privacy and propose the notion of "forward search privacy". Intuitively, it requires search operations over *newly added documents* do not leak any information about past queries. The enhanced security notion poses new challenges to the design of SSE. We address the challenges by developing the hidden pointer technique (HPT) and propose a new SSE scheme called `Khons`, which satisfies our security notion (with the original forward privacy notion) and is also efficient. We implemented `Khons` and our experiment results on large dataset (wikipedia) show that it is more efficient than existing SSE schemes with forward privacy.

Index Terms—Encrypted web application, data privacy, format-preserving encryption, shadow DOM, cloud storage.

1 Introduction

Data storage outsourcing is increasingly prevalent fuelled by the development of cloud computing. While the users enjoy benefits such as low cost and ubiquitous access, data privacy becomes a major concern. To protect data privacy, the users usually encrypt data before uploading it to the untrusted storage server. However, encryption makes data incomprehensible so that common retrieval methods such as the keyword search cannot be directly executed on ciphertexts. To solve this problem, searchable symmetric encryption (SSE) was introduced in 2000 [11]. It allows a client to store encrypted documents on an untrusted server, then to retrieve all documents containing a certain keyword by submitting a token that cryptographically encodes the keyword.

Now, SSE has been widely used in encrypted databases [1–7] and encrypted emails [8]. Take the CryptDB [4] as an example, besides supporting SQL LIKE operator by using an SSE scheme [11] directly, it uses SSE to implement SQL equality queries (=, !=, IN, NOT IN, etc) when the values in the column are not unique. Recently, it has been proposed to apply SSE to support rich queries [5], e.g. conjunctive query [9], range query [12], and so on. Moreover, ARX has applied SSE to provide equality query over the encrypted NoSQL databases.

- *J. Li is with the School of Computer Science, Guangzhou University, the College of Cyber Science, Nankai University, China and the Department of Computer Science, Virginia Polytechnic Institute and State University, USA. Email: lijin@gzhu.edu.cn.*
- *Y. Huang, Y. Wei, S. Lv and Z. Liu are with the College of Cyber Science and the College of Computer Science, Nankai University, China, 300071. Email: onlyerir@163.com, suifengrudao@outlook.com, lv_si_yi@163.com, liuzheli@nankai.edu.cn.*
- *C. Dong is with School of Computing, Newcastle University, Newcastle upon Tyne, UK. Email: changyu.dong@ncl.ac.uk.*
- *W. Lou is with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA. Email: wjlou@vt.edu.*
- *Corresponding author: Zheli Liu (liuzheli@nankai.edu.cn), Jin Li (lijin@gzhu.edu.cn).*

1.1 The Need for Forward Privacy

Deterministic encryption used in SSE makes it easy for the malicious server to observe repeated queries and other information. These leakages are modeled as search pattern (repetitive pattern in search queries), size pattern (the number of search results) [13] and access pattern (how the encrypted data or indexes are accessed). Generally, these leakages could be eliminated by using an oblivious RAM (ORAM) [10, 18, 19, 27, 28]. However, ORAM usually brings heavy computational overhead and bandwidth cost for each keyword search. Thus, a practical SSE has to allow some information leakage in exchange for acceptable efficiency. Unfortunately, these leakages have been abused to attack SSE schemes in different ways [14, 15, 29, 38, 39].

In 2016, Zhang et al. [16] proposed the file-injection attack. This attack assumes that the adversary can inject files, i.e. to craft a set of documents and trick the client into encrypting them. By injecting the carefully selected files, the adversary can recover keywords, which should be kept private, from search tokens submitted by the client. The attack is very effective and requires only a small number of files to be injected. The problem highlighted by this attack is that the security notion widely used in the past is too weak. More specifically, it allows the adversary to gain knowledge about keywords queried in the past by relating past submitted token to newly updated files. The attack calls for a more stringent treatment of information leakage in SSE and makes forward privacy the baseline for newly developed SSE schemes.

Forward privacy (FP) requires that the update (addition and deletion) operations cannot be linked to previous search queries. From a practical point of view, it is important

TABLE 1: Comparison with prior forward private SSE schemes. K is the number of keywords, m is the number of sub keywords, D is the number of documents in EDB. The n_w is the size of the search result set matching keyword w, a_w is the total number of entries matching keyword w, d_w is the number of deleted entries matching w. RT denotes round trip, BP denotes backward-private, PT denotes Partition-based technique, F_uP denotes forward update privacy and F_sP denotes forward search privacy.

Scheme	Computation		Communication			Client Storage	BP	PT	F_uP	F_sP
	Search	Update	Search	RT	Update					
Previous works										
TWORAM [10]	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(\log^2 N)$	$\tilde{O}(a_w \log N + \log^3 N)$	2	$\tilde{O}(\log^3 N)$	$O(1)$	–	–	√	–
Dual [25]	$O(a_w - d_w)$	$O(1)$	$O(n_w)$	1	$O(1)$	$O(K \log D)$	–	–	√	×
Fides [24]	$O(a_w)$	$O(1)$	$O(a_w)$	2	$O(1)$	$O(K \log D)$	II	–	√	×
Diana$_{del}$ [24]	$O(a_w)$	$O(\log a_w)$	$O(n_w + d_w \log a_w)$	2	$O(1)$	$O(K \log D)$	III	–	√	×
Janus [24]	$O(n_w \cdot d_w)$	$O(1)$	$O(n_w)$	1	$O(1)$	$O(K \log D)$	III	–	√	×
This works										
Khons	$O(n_w)$	$O(1)$	$O(n_w)$	2	$O(1)$	$O(m \log D + D \log K)$	III	√	√	√

for users to securely and dynamically build the encrypted database [34–37]. Since 2016 several schemes have been proposed to achieve this goal using different cryptographic primitives, including Sophos [20] (uses trapdoor permutation (TDP)), Diana [24] (uses Constrained Pseudorandom Function (CPRF)), and Dual [25] (uses keyed hash function). Among them, Diana [24] and Dual [25] only use symmetric primitives and are more efficient. Backward privacy (BP) is a related security notion that prevents search operations from leaking the matching elements after they have been deleted. It was showed in [24] that a two-roundtrip backward-private SSE can be obtained from any forward private SSE scheme by applying the generic transformation.

1.2 Security Limitation of Forward Privacy

Currently, FP (and BP, as it can be obtained from an FP scheme) has become a basic security requirement for SSE without ORAM. However, we think that FP is not satisfactory enough, and there is still room for improving security.

The problem of FP is that new updates remain unlinkable to previous search queries only until a search query is performed. The search query links all updates matching to the same keyword. This is the reason why it can resist the adaptive file-injection attack, but not statistical inference attacks [14, 15, 17, 29]. These statistical attacks are based on a large collection of information about the same query behaviors. If the newly added documents remain unlinkable after search queries, it will be more difficult for an adversary to infer the keywords being queried and may defend some statistical attacks.

To some degree, the current FP concerns only information leakage caused by update operations and can be regarded as "*forward update privacy*" (F_uP). A more stringent security notion should also consider information leaked through search operations. If an SSE scheme is forward update privacy and its search operation over newly updated documents or over documents within a period of time does not leak the past query information, it achieves a new security notion. We call it "*forward search privacy*" (F_sP).

1.3 Challenges of designing a forward search privacy scheme

Primarily, we divide forward search privacy into two types: *weak forward search privacy* and *strong forward search privacy* respectively, which will be described detailedly in Section 3.2. Among them, the latter can only be achieved by Oblivious RAM or similar construction. However, ORAM will bring heavy computation and communication overhead, which make SSE impractical. As we focus on practically relevant SSE schemes, strong forward search privacy is beyond the scope of this article.

For the weak forward search privacy, the most critical challenge is to design a brand new SSE scheme which can balance the security and efficiency. For the consideration of efficiency, most of SSE schemes leverage inverted index so that maps a keyword to a set of documents containing this keyword. Conceptually, for each keyword w, there is a list $\mathbb{L}_w$ such that each element in $\mathbb{L}_w$ is a pair (index, ind), where ind is the identifier of a document that contains w, and index is a pointer to the previous (or next) element in $\mathbb{L}_w$. To achieve forward privacy, all the $\mathbb{L}_w$s are merged into a single list $\mathbb{L}$, and some cryptographic primitives are used so that when a new element is added into $\mathbb{L}$, one cannot link it to a specific $\mathbb{L}_w$ (until the next search query for w). A search query for w can be easily answered by giving the index of latest element in $\mathbb{L}_w$, and decrypting the previous element's index one by one to recover all identifiers. However, it is not suitable for the goal of forward search privacy, because it is hard to get a part of elements without leaking which list they belong to and the relation with other elements in the same list. How to achieve the highest possible level of security while preserving the efficiency of SSE can be a huge challenge.

1.4 Our Contributions

Our contributions are summarized as follows.

1) We explain the security limitation of current forward privacy and propose an enhanced notion *forward search privacy*, which ensures that searches over newly added documents do not leak the past query information. We point out that forward search private SSE will leak less information than SSE which only satisfies the original FP notion. We also describe its applications in building secure encrypted applications and improving efficiency in the design of encrypted databases.
2) We design Khons scheme which achieves forward search privacy and supports parallel query. It has

both high security and efficiency. Experiment results show that with the large dataset (wikipedia) and RockDB, in multiple thread environment, `Khons` is at least 3× faster than Dual [25] and 2× than Fides [24].

TABLE 1 summarizes the comparisons between our schemes and prior forward private schemes.

2 PRELIMINARIES

In this section, we will introduce the notations used in this paper, the cryptographic primitives, and the definition of Searchable Symmetric Encryption (SSE).

2.1 Notations

In this paper, $\mathsf{negl}(\lambda)$ is a negligible function, where λ is the security parameter. Unless specified explicitly, the symmetric keys are strings of λ bits, and the key generation algorithm uniformly samples a key in $\{0,1\}^\lambda$. We only consider (probabilistic) algorithms and protocols running in time polynomial in the security parameter λ. In particular, adversaries are probabilistic polynomial-time (PPT) algorithms. For a finite set X, $x \xleftarrow{\$} X$ means that x is sampled uniformly from X.

A database $\mathsf{DB}=(\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^D$ is a tuple of index/keyword-set pair with $\mathsf{ind}_i \in \{0,1\}^\mu$ and $\mathsf{W}_i \subseteq \{0,1\}^*$ where ind_i is a document identifier and W_i is a set of keywords matching document ind_i. The keyword set of the database DB is $\mathbb{W} = \bigcup_{i=1}^D \mathsf{W}_i$ and the document set is $\mathbb{D} = \bigcup_{i=1}^D \{\mathsf{ind}_i\}$. We define the number of search results for keyword w as n_w and the set of documents containing a keyword w as $\mathsf{DB}(w) = (\mathsf{ind}_i | w \in \mathsf{DB}(\mathsf{ind}_i))$ where $|\mathsf{DB}(w)|$ is a_w. A keyword w can be divided into a set of sub keywords $S_w = \{w_i | E_{K_s}(w,i), 1 \le i \le x\}$ where K_s is the encryption key and x is a constant. Let $D = |\mathbb{D}|$ denotes the number of documents in DB, $W = |\mathbb{W}|$ the total number of keywords, and N be the number of document/keyword pairs (we identify documents with their identifier). Note that N can be written as $N = \sum_{i=1}^n |\mathsf{DB}(\mathsf{ind}_i)| = \sum_{w \in \mathbb{W}} |\mathsf{DB}(w)|$.

2.2 Cryptographic primitives.

A private-key encryption scheme [26, 30] is a set of three polynomial-time algorithms $\mathsf{SK} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ where Gen is a probabilistic algorithm that takes as a input a security parameter λ and returns a secret key K_s, Enc is a probabilistic algorithm takes as inputs a key K_s and a message m and returns a ciphertext c and Dec is a deterministic algorithm that takes as inputs a key K_s and a ciphertext c and returns m if K_s was the key under which c was encrypted. Informally, a private-key encryption scheme is CPA-secure if for any probabilistic polynomial-time adversary $\mathcal{A}$, there exists a negligible function negl such that

$$\Pr[\mathsf{PrivK}^{\mathsf{CPA}}_{\mathcal{A},\mathsf{SK}}(\lambda) = 1] \leqslant \frac{1}{2} + \mathsf{negl}(\lambda)$$

For encryption schemes, we employ pseudo-random functions (PRF), which is a polynomial-time computable functions. PRF cannot be distinguished from random functions by any probabilistic polynomial-time adversary. A hash function is a pair of probabilistic polynomial-time algorithms (Gen, H) where Gen is a probabilistic algorithm which takes as input a security parameter 1^n and outputs a key s. We assume that 1^n is included in s. There exists a polynomial l such that H is (deterministic) polynomial-time algorithm that takes as input a key s and any string $x \in \{0,1\}^*$, and outputs a string $H^s(x) \in \{0,1\}^{l(n)}$. It is not much more difficult to see that a random oracle acts like a hash function.The success probability of any polynomial-time adversary $\mathcal{A}$ in the following game is negligible:

- A random function H is chosen.
- $\mathcal{A}$ succeeds if it outputs x, x' with $H(x) = H(x')$ but $x \neq x'$.

We refer the reader to [30] for formal definitions of CPA-security, PRFs and Hash functions.

2.3 Searchable Symmetric Encryption

Initially, SSE is proposed to protect static data and thus does not support update operations. Recently, most research focuses on constructing dynamic searchable symmetric encryption (DSSE) [6, 21–23] that offers search capability and allows dynamically adding and deleting documents. We review the definition of dynamic SSE in [20]. A *DSSE* scheme Π=(Setup, Search, Update) contains a Setup algorithm, and two protocols Search and Update:

- $\mathsf{Setup}(\mathsf{DB}) \rightarrow (\mathsf{EDB}, sk, \sigma)$ is an algorithm for setting up the encrypted database supporting keyword search. It takes as input a database DB and outputs (EDB, sk, σ) where EDB is the encrypted database, sk is a secret key, and σ is the client's state.
- $\mathsf{Search}(sk, q, \sigma;\ \mathsf{EDB}) = (\mathsf{Search}_C(sk, q, \sigma), \mathsf{Search}_S(\mathsf{EDB}))$ is a client-server protocol supporting search operation of a document. The client takes as inputs the key sk, its state σ, and a search query q. The server takes as input EDB , outputs the results as document identifiers matching the query q.
- $\mathsf{Update}(sk, \sigma, \mathsf{op}, \mathsf{in};\ \mathsf{EDB}) = (\mathsf{Update}_C(sk, \sigma, \mathsf{op}, \mathsf{in}), \mathsf{Update}_S(\mathsf{EDB}))$ is a client-server protocol supporting update operation of a document. The client takes as inputs the key sk, an operator op which is taken from the set $\{\mathsf{add}, \mathsf{del}\}$, client's state σ and an input in parsed as the document ind and a set W of keywords. The server takes as input the EDB.

Adaptive Security of SSE. The standard security definition of a DSSE scheme follows the ideal/real simulation paradigm [20, 24]. It requires the server to know as little as possible about the content of database and queries. More specifically, we wish the adversary will learn nothing except for some obvious leakages. We use a stateful leakage functions to express the information leaked to the adversary by each SSE operation, which is $\mathcal{L} = (\mathcal{L}^{\mathsf{Setup}}, \mathcal{L}^{\mathsf{Search}}, \mathcal{L}^{\mathsf{Update}})$, whose components correspond respectively to the information leaked to the adversary by Setup, Search and Update operations. The definition ensures that the scheme will reveal no information beyond what is inferred from the leakage functions.

The adversary aims to distinguish between a real world SSEREAL and an ideal world SSEIDEAL. In these worlds, the adversary can trigger Setup, Search and Update operations with parameters which are chosen by herself. Then, she can observe the execution of the scheme like what the server does. We describe what the adversary $\mathcal{A}$ does in real world and ideal world specifically as follows.

- In the SSEREAL world, the DSSE scheme is executed honestly. The adversary $\mathcal{A}$ chooses a database DB. The experiment runs Setup(DB) and returns EDB to $\mathcal{A}$. Then, $\mathcal{A}$ adaptively chooses queries q_i. The experiment runs Search(sk, q_i, σ_i; EDB_i) or Update(sk, σ_i, op, in_i; EDB_i) depending on the protocol of query q_i and returns (σ_{i+1},DB(w_i) EDB_{i+1}) or (σ_{i+1}, EDB_{i+1}). Finally, the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.
- In the SSEIDEAL world, the adversary sees messages generated by a PPT algorithm $\mathcal{S}$, known as the *simulator*, that has access to only the leakage functions but not the database or queries. The adversary $\mathcal{A}$ chooses a database DB. The simulator returns an encryption database $\mathsf{EDB} \leftarrow \mathcal{S}(\mathcal{L}^{\mathsf{Setup}}(\mathsf{DB}))$ to $\mathcal{A}$. Then, $\mathcal{A}$ adaptively chooses queries q_i. The experiment runs $\mathcal{S}(\mathcal{L}^{\mathsf{Search}}(q_i))$ or $\mathcal{S}(\mathcal{L}^{\mathsf{Update}}(q_i))$ to answer the query q_i. Finally, the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.

If an adversary can distinguish the real game and the ideal game of DSSE with only a negligible probability, we say that DSSE achieves adaptive security, which is defined as follows.

Definition 1. (Adaptive security). A DSSE scheme Π with a collection of leakage functions $\mathcal{L}$ is *$\mathcal{L}$-adaptively-secure*, if for any polynomial-time adversary $\mathcal{A}$ issuing a polynomial number of queries $q(\lambda)$, there exists a PPT simulator S such that

$$|\mathsf{Pr}[\mathrm{SSEREAL}^{\Pi}_{\mathcal{A}}(\lambda, q)=1]\text{-}\mathsf{Pr}[\mathrm{SSEIDEAL}_{S,\mathcal{A},\mathcal{L}}(\lambda, q)=1]| \leq \mathsf{negl}(\lambda)$$

3 PRIVACY DEFINITION

We now review the definitions of forward privacy and backward privacy and define the new forward search privacy. The following notations are used throughout the paper.

The repetition of token (i.e., queried keywords) sent to the server will be leaked in the most SSE schemes. If this leakage is limited to search queries, we call it *search pattern*. If this leakage includes the repetition of updated keywords, we call it the *query pattern*.

The leakage function $\mathcal{L}$ will keep as state the *query list* Q:the list of all queries issued so far, and whose entries are (i, w) for a search query on keyword w, or $(i, \mathsf{op}, \mathsf{in})$ for an op update query with input in. The integer i is a timestamp, initially sets to 0, and which is incremented at each query. Let sp(x) and qp(x) denote the search and query patterns respectively which are defined as

$$\mathsf{sp}(x) = \{j : (j, x) \in Q\} \text{(only matches search queries)}$$

$$\mathsf{qp}(x) = \{j : (j, x) \in Q\} \text{ or } (j, \mathsf{op}, \mathsf{in}) \in Q \text{ and } x \text{ appears in } \mathsf{in}$$

In this paper, $\mathsf{TimeDB}(w)$ is the list of all documents matching w, excluding the deleted ones, together with the timestamp of when they were inserted in the database. $\mathsf{Updates}(w)$ is the list of timestamps of updates on w. Deletion history $\mathsf{DelHist}(w)$ is the list of timestamps for all deletion operations, together with the timestamp of the inserted entry it removes.

3.1 Forward Update Privacy

The traditional forward privacy [20] is that the server cannot learn whether the newly updated documents match a previously searched keyword or not. In this paper, we define it as forward update privacy.

Definition 2. (Forward update privacy). A *$\mathcal{L}$-adaptively-secure* SSE scheme is forward-update-private iff the update leakage function $\mathcal{L}^{\mathsf{Update}}$ can be written as

$$\mathcal{L}^{\mathsf{Update}}(\mathsf{op}, \mathsf{ind}, W) = \mathcal{L}'(\mathsf{ind}, |W|)$$

where ind denotes the identifiers of the newly added documents, $|W|$ denotes the number of keywords of the newly added document and $\mathcal{L}'$ is stateless.

As shown in Definition 2, forward update privacy requires that the information leaked in update operation should not be more than the identifier and the number of keywords of newly updated document.

3.2 Forward Search Privacy

In existing SSE schemes, a search token leaks a significant amount of information. This is captured by the leakage function $\mathcal{L}^{\mathsf{Search}}(w) = \mathcal{L}'(\mathsf{TimeDB}(w))$, where $\mathcal{L}'$ is stateless.

Forward search privacy is defined on the basis of forward update privacy. It further prevents the server to know whether a search over newly updated documents matches a previously searched keyword. We first introduced the notion of strong forward search privacy. An SSE scheme satisfies strong forward search privacy if the search token leaks no information. We define it as follows:

Definition 3. (Strong forward search privacy). A *$\mathcal{L}$-adaptive-secure* SSE scheme is strong forward-search-private, iff functions $\mathcal{L}^{\mathsf{Search}}$ can be written as:

$$\mathcal{L}^{\mathsf{Search}}(w) = \mathcal{L}'(\perp)$$

where $\mathcal{L}'$ is stateless.

This is a very strong notion, but on the other hand is also very difficult to achieve. In fact, this implies the search operation is fully oblivious, and cannot be achieved unless expensive protocols such as ORAM or PIR are used. For practicality, we also define a weaker notion of forward search privacy that leaks partial pattern:

Definition 4. (Weak forward search privacy). Let $S_w = \{w_1, ..., w_x\}$ denote a set of sub keywords for a keyword w where x is a constant. A *$\mathcal{L}$-adaptive-secure* SSE scheme is weak forward-search-private, iff the leakage functions $\mathcal{L}^{\mathsf{Search}}$ can be written as:

$$\mathcal{L}^{\mathsf{Search}}(w_i) = \mathcal{L}'(\mathsf{TimeDB}(w_i))$$

where $\mathcal{L}'$ is stateless and $|\mathsf{TimeDB}(w_i)| = a_w$ for a_w is a constant.

3.3 Backward Privacy

An SSE scheme satisfies backward privacy if after deleting a document ind matching keyword w, the server cannot reveal the deleted document ind from the subsequent search of keyword w.

In 2017, Bost et al. [24] have defined backward privacy at three levels: BP-I, BP-II and BP-III. They all leak the documents currently matching w, when they were inserted. As for other leakages: BP-I only allows the leakage of "the total number of updates on w"; BP-II further allows the leakage of "when all the updates on w happened"; and BP-III further allows the leakage of "which deletion update canceled which insertion update". We review these definitions as follows.

Definition 5. (BP-I). *A $\mathcal{L}$-adaptively-secure SSE scheme is* insertion pattern revealing backward-private *iff leakage functions* $\mathcal{L}^{\mathsf{Search}}$ can be written as:

$$\mathcal{L}^{\mathsf{Update}}(\mathsf{op}, w, \mathsf{ind}) = \mathcal{L}^{'}(\mathsf{op}),$$

$$\mathcal{L}^{\mathsf{Search}}(w) = \mathcal{L}^{''}(\mathsf{TimeDB}(w)),$$

where $\mathcal{L}^{'}$ and $\mathcal{L}^{''}$ are stateless and $|\mathsf{TimeDB}(w)| = a_w$ for a_w is a constant.

Definition 6. (BP-II). *A $\mathcal{L}$-adaptively-secure SSE scheme is* update pattern revealing backward-private *iff leakage functions* $\mathcal{L}^{\mathsf{Search}}$ can be written as:

$$\mathcal{L}^{\mathsf{Update}}(\mathsf{op}, w, \mathsf{ind}) = \mathcal{L}^{'}(\mathsf{op}, w),$$

$$\mathcal{L}^{\mathsf{Search}}(w) = \mathcal{L}^{''}(\mathsf{TimeDB}(w), \mathsf{Updates}(w)),$$

where $\mathcal{L}^{'}$ and $\mathcal{L}^{''}$ are stateless and $|\mathsf{TimeDB}(w)| = a_w$ for a_w is a constant.

Definition 7. (BP-III). *A $\mathcal{L}$-adaptively-secure SSE scheme is* weakly backward-private iff leakage functions $\mathcal{L}^{\mathsf{Search}}$ can be written as:

$$\mathcal{L}^{\mathsf{Update}}(\mathsf{op}, w, \mathsf{ind}) = \mathcal{L}^{'}(\mathsf{op},\ w),$$

$$\mathcal{L}^{\mathsf{Search}}(w) = \mathcal{L}^{''}(\mathsf{TimeDB}(w), \mathsf{DelHist}(w)),$$

where $\mathcal{L}^{'}$ and $\mathcal{L}^{''}$ are stateless and $|\mathsf{TimeDB}(w)| = a_w$ for a_w is a constant.

The difference between weak forward search and backward privacy is that weak forward search privacy only leaks partial query pattern. Note that in the defintion 4, the leakage function is based on $\mathsf{TimeDB}(w_i)$ where w_i is a sub-keyword of keyword w, while in definition 5 -7, the leakage function is based on $\mathsf{TimeDB}(w)$, i.e. it exposes the whole query pattern.

4 Overview of Techniques

In this section, we will introduce two techniques to help achieve the forward search privacy and then describe a toy construction.

4.1 Partitioning Technique

In SSE schemes, indexes are used widely. In our construction, inverted index is used to facilitate search queries in the form of a pair (key, value) , where key is a keyword and value is a list of identifiers of documents containing this keyword. Given a keyword, we can retrieve all the documents that contains the keyword efficiently.

We partition the inverted index into disjoint partitions and generate a sub-keyword for each partition to reduce information leakage in SSE. In this way, a search token of a keyword will become multiple search tokens, each for a different partition. More specifically, we add the identifier of the document to a partition using a sub-keyword derived from w as the key when adding a document that contains a keyword w. When performing a search query, we allow the client to submit a search token of a sub-keyword to search over a subset of documents in this partition. If we set only one partition for a keyword, it will be the traditional inverted index.

4.2 Hidden Pointer Technique (HPT)

To use the partitioning technique in SSE, we need to build encrypted lists so that we can store all indexes at the server securely.

We first define the data structure. A data block is a four-tuple (id, data, key, ptr), where id is the block identifier, data is a piece of data, key and ptr are the encryption key and identifier of another block (suffix block). If a block has no suffix block, key is set to $\perp$. In a data block b, data, ptr and key fields should be encrypted. We denote $b.id$ as the id of block b and $b.\mathsf{value}$ as all the other contents of b including $b.\mathsf{data}$, $b.\mathsf{key}$ and $b.\mathsf{ptr}$.

As shown in Fig. 1, HPT allows us to add data blocks into an encrypted linked list. Let $\mathbb{L}$ be a list of data blocks. Let the *head block* be the latest block being added to $\mathbb{L}$ and the *tail block* be the oldest block in $\mathbb{L}$. We can describe how HPT works by the following algorithms:

- $\mathsf{AddHead}(\mathbb{L}, id, \mathsf{value}, 1^{\lambda})$: it adds a new block as the *head block* to list $\mathbb{L}$. It has four steps: 1) generate a data block as b=(id, data, $\mathbb{L}.\mathsf{head}.\mathsf{key}$, $\mathbb{L}.\mathsf{head}.id$); 2) sample a random key k from $\{0,1\}^{\lambda}$; 3) use k to encrypt the $b.\mathsf{value}$; 4) add b to $\mathbb{L}$.
- $\mathsf{RetrieveABlock}(\mathbb{L}, id, k)$: it retrieves a data block from the list $\mathbb{L}$. It has three steps: 1) find block b by identifier id; 2) decrypt block $b.\mathsf{value}$ by the corresponding key k; 3) return b.
- $\mathsf{RetrieveList}(\mathbb{L}, id, k)$: it retrieves all data blocks from a sublist of $\mathbb{L}$, by calling $\mathsf{RetrieveABlock}(\mathbb{L}, id, k)$ repeatedly until the *tail block* (i.e. a block b such that $b.\mathsf{key} = \perp$) is visited.

We can use it to build secure index. For example, we can build an inverted index $(w, \mathbb{L}_w)$ such that $\mathbb{L}_w$ is a list built using HPT. The client can keep the head of the list and store $\mathbb{L}_w$ on the server. The list $\mathbb{L}_w$ can be updated by the client by adding a new block, and the client can search the index by revealing the id of the head block and the encryption key to the server. Similarly, we can also build a forward index using HPT that maps a document identifier to a list of keywords contained in the document.

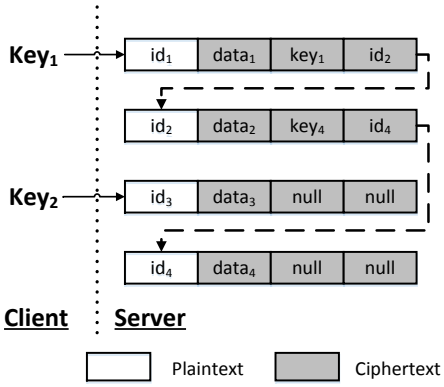


Fig. 1: An example of HPT. There are four data blocks, whose identifiers and encryption keys are (id_1, key_1), (id_2, key_2), (id_3, key_3) and (id_4, key_4). The *head block* in the target list is the id_1, whose encryption key is maintained in the client; the id_2 is an *inner block* and its key is stored in the prefix block id_1; the *tail block* is the id_4, whose encryption key is stored in the prefix block id_2. The *ptr* value of id_4 is $\perp$ because it is the end of this list.

One advantage of HPT is that one can have multiple lists and store their blocks in arbitrary order, and can still retrieve each individual list correctly by the id and the encryption key of the head block of the list. Another advantage is that if the server stores multiple lists and a new block comes in, the server will not be able to tell which list this block belongs to (until later the user reveals it), which is important if we want to achieve forward privacy.

4.3 A Toy Construction

We propose a toy construction, which is shown in Fig. 2, to achieve forward search privacy. For a keyword w, we divide it into a set of sub-keywords S_w=$\{w_1, \cdots, w_x\}$ according to the total number of associated documents and the number of documents in a partition where x is a constant. Each sub-keyword w_i maintains its own index list $\mathbb{L}_{w_i}$. Each $\mathbb{L}_{w_i}$ can be retrieved using the sub-keyword state that includes the id and encryption key of its head node. The total number of partitions and sub-keyword states should be maintained at the client.

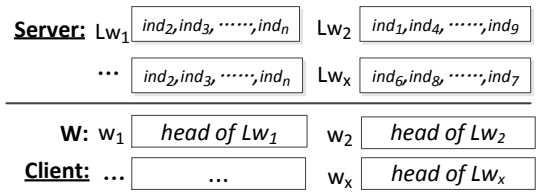


Fig. 2: The toy construction of forward search privacy

As shown in Fig. 2, the server stores $\mathbb{L}_{w_i}$ ($i \in [1, x]$) and the client stores the head of $\mathbb{L}_{w_i}$. Thus, for each keyword, the client maintains at most x heads. With these heads, the client can search each partition.

Challenges. A traditional SSE can be used to implement the single partition search query by issuing a search token of the associated sub-keyword. However, to implement a complete search query that covers all partitions, it must issue all the search tokens of all sub-keywords.

Another huge challenge is *immediate deletion* problem. Immediate deletion means that when a document is deleted, it should immediately delete the related keyword-document pairs in the inverted index. If the final SSE does not support immediate deletion, a single partition query may return documents that have already been deleted.

How to achieve weak forward search privacy. In order to satisfy the requirement of weak $\mathrm{F}_s\mathrm{P}$, when adding a document, the keyword-document pair should not be added into the list that has been searched. If the latest partition has been searched, even if it is not full, we must create a new partition and generate a new sub-keyword for this partition. In other cases, we can add the keyword-document pair into the latest partition directly until it is full.

5 Khons: First SSE Scheme With Forward Search Privacy

In this section, we propose a forward and backward secure SSE scheme named "`Khons`". It satisfies the weak forward search privacy and forward update privacy (thus can be extended to backward privacy(BP-II)). We combine inverted indexes and forward indexes for efficient search, addition and more importantly immediate deletion.

5.1 Storage Structure

The server stores the data blocks in a dictionary $\mathbb{D}$ where $\mathbb{D}[id]$ stores a data block with identifier id. As in section 4.3, from each keyword w we derive a set of sub-keywords $S_w = \{w_i | E_{K_s}(w, i), 1 \leq i \leq x\}$ where K_s is the encryption key. For each sub-keyword w_i, we build a list $\mathbb{L}_{w_i}$ for the single partition query which is treated as an inverted index.

To support the full search query, for each keyword w, we made some change to the tail blocks of the lists. In a single partition search, the search is restricted to one partition because the server knows only the head id and encryption key for that one list (partition). When reaching the tail block of the list, the search has to stop because there is not pointer to the next block. Now in the list $\mathbb{L}_{w_i}$, we store the head head id and encryption key, both encrypted, for $\mathbb{L}_{w_{i-1}}$. More specifically, the encrypted head id is stored in the data field and the encrypted encryption key is stored in the key field. Encrypting the two pieces of information ensures that the server cannot do a full search without the user's permission. The encryption brings a problem, that is, where to store the keys. To solve this problem, we also build another list $\mathbb{L}_w$. It has one block for each $\mathbb{L}_{w_i}$ that store the key to decrypt data stored in the tail block of $\mathbb{L}_{w_i}$.

To support immediate deletion, for each document ind, there is a list $\mathbb{L}_{\mathsf{ind}}$. $\mathbb{L}_{\mathsf{ind}}$ play a role as forward index.

The client stores key K_s and map $\mathbf{M}_b$, $\mathbf{M}_b{}^*$, $\mathbf{M}_f$. Key K_s is the user secret key for generating a one-time key to encrypt data. Additionally, we apply a map $\mathbf{M}_b$, $\mathbf{M}_b{}^*$ to store the state of each keyword and sub-keyword. $\mathbf{M}_f$ stores the state of each document. For each keyword w, we define $\mathbf{M}_b[w]$ stores (num_p, cnt_p, key, flag) with initial value of (1,0,$\perp$, *false*), where num_p is the total number of partitions, cnt_p, key_w the number of blocks and the encryption key of the tail block in the latest (num_p-th) partition and flag

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2894411, IEEE Transactions on Dependable and Secure Computing

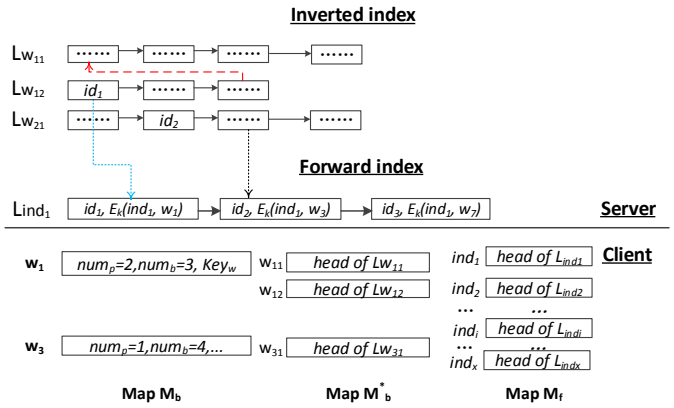


Fig. 3: The storage structure of Khons. There are three lists. List $\mathbb{L}_{\mathsf{ind}_1}$ contains the keywords in document ind_1 in form of (ind_1, w). List $\mathbb{L}_{w_{11}}$ and $\mathbb{L}_{w_{12}}$ contain the document identifiers of the first and second partition of keyword w_1 respectively; however, instead of storing a document identifier ind directly, they store an identifier of a block whose value is (ind, w_1) in list of $\mathbb{L}_{\mathsf{ind}}$. The blue dotted line shows it. List $\mathbb{L}_{w_1}$ builds a hidden list among lists of its sub-keywords, by linking all the tail blocks in them. The red line shows it.

denotes whether the latest (num_p-th) partition is accessed. Notice that a partition can only hold at most P blocks. For each sub-keyword w_i, $\mathbf{M}_b{}^*[w_i]$=(id, key, cnt, flag) with initial value of ($\perp$,$\perp$,0,*false*), where id, key is the identify and encryption key of the head block in $\mathbb{L}_{w_i}$, cnt the number of blocks in this partition related to w_i and flag denotes whether this partition (keyword w_i) is accessed. For each document ind, we adopt $\mathbf{M}_f$[ind]=(id, key), i.e., the pointer information of head block of list $\mathbb{L}_{\mathsf{ind}}$. Fig. 3 shows the details of the storage structure.

5.2 Basic Algorithm

In the following pseudo codes, encryption $E_k(m)$ and decryption $D_k(c)$ are implemented by an IND-CPA (indistinguishability against the chosen plaintext attack) secure symmetric cryptographic primitive with the encryption key k. H, H_1, H_2 and H_3 are keyed hash functions.

Algorithm 1 Khons.**Setup**()

1: $K_s \xleftarrow{\$} \{0,1\}^\lambda, \mathsf{key}_h \xleftarrow{\$} \{0,1\}^\lambda$
2: $\mathbf{M}_b, \mathbf{M}_b{}^*, \mathbf{M}_f \leftarrow$ empty map
3: $\mathbb{D} \leftarrow$ empty dictionary

Khons.**Setup.** The client randomly generates K_s as user key and initializes map $\mathbf{M}_b$, $\mathbf{M}_b{}^*$ and $\mathbf{M}_f$ for maintaining the pointer information of each keyword and document. The server initializes the dictionary $\mathbb{D}$ to store data blocks.

Khons.**Add**. To add a document (with identifier ind) matching w, there are three steps. Khons firstly inserts a block b with value of (ind, w) into list $\mathbb{L}_{\mathsf{ind}}$. Then, it gets the keyword state stored in $\mathbf{M}_b[w]$ and obtains sub-keyword w_i of the latest partition. Notice that, if the latest partition has been accessed or is full (P is maximum number of documents in a partition), it must create a new partition by adding a new sub-keyword w_i for w. If the number of elements in the latest partition is less than P, we must fill with dummy blocks until the number of elements in this partition reaches P. Finally, a block with the value of $b.id$ will be inverted into list $\mathbb{L}_{w_i}$.

We explain how to add the first block to list $\mathbb{L}_{w_i}$ in more details. To build list $\mathbb{L}_w$, we firstly add a special tail block B_{tail} into list $\mathbb{L}_{w_i}$. The block B_{tail} stores the identify and encryption key of the head block of list $\mathbb{L}_{w_{i-1}}$ and encryption key of the tail block of list $\mathbb{L}_{w_{i-1}}$. The prefix block of B_{tail} in list $\mathbb{L}_{w_i}$ only stores the identifier of B_{tail} but does not store its encryption key. The details are shown in Algorithm 2.

Khons.**Delete**. To delete a document with identifier ind, the client gets pointer information of the list associated with it from map $\mathbf{M}_f$, i.e., (id, key) $\leftarrow$ ($\mathbf{M}_f$[ind].id, $\mathbf{M}_f$[ind].key), and sends them to the server. Then, the server repeatedly gets all the blocks in list $\mathbb{L}_{\mathsf{ind}}$, deletes them (marking them as inaccessible) and returns their identifiers back to the client for future reuse. The details are shown in Algorithm 3.

Notice that Khons.**Delete** only influences the blocks in $\mathbb{L}_{\mathsf{ind}}$. The identifier ind is not stored in $\mathbb{L}_w$ or $\mathbb{L}_{w_i}$ directly,

Algorithm 2 Khons.**Update**(**add**, w, ind, σ; EDB)

Client:
1: $(id_f, \mathsf{key}_f) \leftarrow (\mathbf{M}_f[\mathsf{ind}].id, \mathbf{M}_f[\mathsf{ind}].\mathsf{key})$

forward index
2: $id_f^* \xleftarrow{\$} \{0,1\}^\lambda, \mathsf{key} \xleftarrow{\$} \{0,1\}^\lambda$
3: $\mathsf{mask} \leftarrow H_1(\mathsf{key}, id_f^*)$
4: $\mathsf{value} \leftarrow E_{K_s}(\mathsf{ind}\|w)\|\mathsf{key}_f\|id_f$
5: $(b.id, b.\mathsf{value}) \leftarrow (id_f^*, \mathsf{value} \oplus \mathsf{mask})$
6: $\mathbf{M}_f[\mathsf{ind}].id \leftarrow id_f^*, \mathbf{M}_f[\mathsf{ind}].\mathsf{key} \leftarrow \mathsf{key}$

inverted index
7: $(\mathsf{num}_p, \mathsf{cnt}_p, \mathsf{key}_w) \leftarrow (\mathbf{M}_b[w].\mathsf{num}_p, \mathbf{M}_b[w].\mathsf{cnt}_p, \mathbf{M}_b[w].\mathsf{key}_w)$
8: $w_i \leftarrow H(\mathsf{key}_h, w\|\mathsf{num}_p)$
9: $(id_b, \mathsf{key}_b) \leftarrow (\mathbf{M}_b^*[w_i].id, \mathbf{M}_b^*[w_i].\mathsf{key})$
10: $(\mathsf{cnt}_b, \mathsf{flag}_b) \leftarrow (\mathbf{M}_b^*[w_i].\mathsf{cnt}, \mathbf{M}_b^*[w_i].\mathsf{flag})$
11: **IF** ($\mathsf{cnt}_p = P \parallel \mathsf{flag}_b = \mathrm{true}$)
12: **IF** ($\mathsf{flag}_b = \mathrm{true}$)
13: padding $P - \mathsf{cnt}_p$ dummy blocks to w_i
14: $\mathsf{num}_p \leftarrow \mathsf{num}_p + 1,\ \mathsf{cnt}_p \leftarrow 1$
15: $w_i \leftarrow H(\mathsf{key}_h, w\|\mathsf{num}_p)$
16: initialize $\mathbf{M}_b^*[w_i]$
17: **ELSE**
18: $\mathsf{cnt}_p \leftarrow \mathsf{cnt}_p + 1$
19: $id_b^* \xleftarrow{\$} \{0,1\}^\lambda,\ \mathsf{key}^* \xleftarrow{\$} \{0,1\}^\lambda$
20: $\mathsf{mask}_2 \leftarrow H_2(\mathsf{key}^*, id_b^*)$
21: **IF** ($\mathsf{cnt}_b = 0$) // Add the first block into the list
22: $id_t \xleftarrow{\$} \{0,1\}^\lambda, \mathsf{key}_t \xleftarrow{\$} \{0,1\}^\lambda$
23: $\mathsf{mask}_3 \leftarrow H_3(\mathsf{key}_t, w)$
24: $B_{\mathsf{tail}}.id \leftarrow id_t$
25: $w_{i-1} \leftarrow H(\mathsf{key}_h, w\|\mathsf{num}_p - 1)$
26: $B_{\mathsf{tail}}.\mathsf{value} \leftarrow (\mathsf{key}_w\|\mathbf{M}_b^*[w_{i-1}].\mathsf{key}\|\mathbf{M}_b^*[w_{i-1}].id) \oplus \mathsf{mask}_3$
27: $id_b \leftarrow id_t,\ \mathsf{key}_b \leftarrow \perp,\ \mathbf{M}_b[w].\mathsf{key}_w \leftarrow \mathsf{key}_t$
28: $(b^*.id, b^*.\mathsf{value}) \leftarrow (id_b^*, (id_f^*\|\mathsf{key}_b\|id_b) \oplus \mathsf{mask}_2)$
29: $\mathbf{M}_b^*[w_i].id \leftarrow id_b^*,\ \mathbf{M}_b^*[w_i].\mathsf{key} \leftarrow \mathsf{key}^*,\ \mathbf{M}_b^*[w_i].\mathsf{cnt}$++
30: Send block b, b^* and B_{tail} (*if exists*) to the server.

Server:
31: $\mathbb{D}[b.id] = b.\mathsf{value}$
32: $\mathbb{D}[b^*.id] = b^*.\mathsf{value}$
33: **IF**(B_{tail} exists)
34: $\mathbb{D}[B_{\mathsf{tail}}.id] = B_{\mathsf{tail}}.\mathsf{value}$

Algorithm 3 Khons.**Update**(**delete**, ind, σ; EDB)

Client:
1: $(id, \mathsf{key}) \leftarrow (\mathbf{M}_f[\mathsf{ind}].id, \mathbf{M}_f[\mathsf{ind}].\mathsf{key})$
2: Send (id, key) to the server.

Server:
3: **REPEAT**
4: $b \leftarrow \mathbb{D}[id]$
5: Delete the block b from the dictionary $\mathbb{D}$
6: $\mathsf{mask} \leftarrow H_1(\mathsf{key}, id)$
7: $b.\mathsf{value} \leftarrow b.\mathsf{value} \oplus \mathsf{mask}$
8: $(id, \mathsf{key}) \leftarrow (b.id_f, b.\mathsf{key}_f)$
9: **UNTIL** $(id = \perp)$

but only stored in $\mathbb{L}_{\mathsf{ind}}$. Because $\mathbb{L}_w$ and $\mathbb{L}_{w_i}$ only store the pointer information to $\mathbb{L}_{\mathsf{ind}}$, the deletion of $\mathbb{L}_{\mathsf{ind}}$ will make the search meaningless.

Algorithm 4 Khons.**Search**(w, i, σ; EDB)

Client:
1: $w_i \leftarrow H(\mathsf{key}_h, w||i)$
2: $(id, \mathsf{key}) \leftarrow (\mathbf{M}_b^*[w_i].id, \mathbf{M}_b^*[w_i].\mathsf{key})$
3: $\mathbf{M}_b^*[w_i].\mathsf{flag} \leftarrow \mathsf{true}$
4: Send token (id, key) to the server.

Server:
5: $\mathbf{S} \leftarrow$ empty set, $j \leftarrow 0$
6: **REPEAT**
7: $b \leftarrow \mathbb{D}[id]$
8: $\mathsf{mask}_2 \leftarrow H_2(\mathsf{key}, id)$
9: $b.\mathsf{value} \leftarrow b.\mathsf{value} \oplus \mathsf{mask}_2$
10: $\mathbf{S} = \mathbf{S} \cup \mathbb{D}[b.id_f]$
11: $(id, \mathsf{key}) \leftarrow (b.id_b, b.\mathsf{key})$
12: **IF** $(\mathsf{key} = \perp)$
13: $id \leftarrow \perp$
14: **UNTIL** $(id = \perp)$
15: Send $\mathbf{S}$ to the Client.

Client:
16: $\mathbf{S} \leftarrow \mathsf{Decrypt}_{K_s}(\mathbf{S})$

Khons.**Search**. Two kinds of query can be supported with the same search token $t = (id, \mathsf{key}, \mathsf{key}_w)$. For a single partition query, such as the i-th partition, the client issues token=($\mathbf{M}_b^*[w_i].id$, $\mathbf{M}_b^*[w_i].\mathsf{key}$). The query in a partition will result in the update of its sub-keyword state, which is aimed to achieve the F$_s$P. Note that each query will return a fixed number of elements due to the padding of dummy blocks in Khons.**Upadte**. Algorithm 4 shows the details of the search operation. We emphasize that every touched blocks will be deleted after querying and updated to corresponding keyword.

Khons achieves backward privacy with two round trips. After receiving the search token, the server retrieves data from either a single partition or all the partitions. For each block b in the list associated with the keyword w, it contains a block identifier. With this block identifier, the server can access the corresponding block in the list associated with a document and get its value (the encrypted document identifier and keyword). Then, the server returns all the encrypted information to the client. The client finally decrypts them, removes the element whose keyword is not w and downloads the documents from the server. The remove operation is caused by the immediate deletion of blocks. If the decrypted keyword of an element is not equal to w, it will be removed from the search results.

5.3 Comparison

The highlight of our index is to support the partial query and maintain the complete query at the same time with the help of HPT technique. The key to implementing the partial query is that encryption key and pointer should be separate. But in [21], each item in index list $\mathbb{L}_{w_i}$ shares the same key, so that partial query is difficult to implement.

There exist some differences among our scheme, [21] and [25] even we all employ forward and backward indexes simultaneously. Notice that the order of the pointers of elements in Khons is the opposite of [21] and [25]. Namely the new file will point to the old file, so that adding a new file no longer require changing the pointing of the previous file. The reverse of pointers will simplify the addition operation of Khons. But in [21] the add operation needs to homomorphically set the previous node's "next" pointers to point to newly added nodes, which brings more computation overhead than our solution. And [25] utilizes label and count to "concatenate" each block corresponding to the same keyword. In add operation of [25], the client will generate and upload a set contains label and count to the dictionary in the server.

For deletion operation, Khons only deletes the elements of file physically according to the linked list in forward index, so that the inverted index will not point to the element in forward index any more. The inverted index will be updated after querying. The deletion of [21] is a "dual operation" to addition. The server will "free" the correlative positions in deletion array and search array. While"freeing" the positions in the search array, it will also homomorphically update the pointers of previous entries in the corresponding keyword list. The dual operation is complicated to update both deletion array and search array. For [25], the server calculates label with key and count and then deletes the corresponding document identifier. The server repeats this process by incrementing the counter until all documents in forward index are deleted. Accordingly, the corresponding entries in inverted index will be deleted. Simultaneous updating of two index increases the cost of deletion operations. Additionally, since Fides[24], Diana$_{\mathsf{del}}$[24] and Janus[24] only unitize inverted index but not forward index, the deletion operation is slightly different. Fides only deletes entries logically. And in Diana$_{\mathsf{del}}$ and Janus, the server maintains two instances of the construction, one for insertions and one for deletions.

5.4 Analysis

In Khons, all the states of documents, keywords and sub-keywords are stored in the client. Thus its client storage overhead is $O(m \log D + D \log K)$, where m, D and K denote the number of sub-keywords, documents and keywords respectively. Khons supports parallel query which is more efficient when performing full query. Because the server can get the pointer information of head block of each partition by retrieving the list $\mathbb{L}_w$.

Setup():
1: $K_s \xleftarrow{\$} \{0,1\}^\lambda, \mathsf{key}_h \xleftarrow{\$} \{0,1\}^\lambda$
2: $\mathbf{M}_b, \mathbf{M}_b{}^*, \mathbf{M}_f \leftarrow$ empty map
3: $\mathbb{D} \leftarrow$ empty dictionary

Update(**add**, w, ind, σ, EDB)
Client:
1: $(id_f, \mathsf{key}_f) \leftarrow (\mathbf{M}_f[\mathsf{ind}].id, \mathbf{M}_f[\mathsf{ind}].\mathsf{key})$
2: $id_f^* \xleftarrow{\$} \{0,1\}^\lambda, \mathsf{key} \xleftarrow{\$} \{0,1\}^\lambda$
3: $\mathsf{mask} \xleftarrow{\$} \{0,1\}^\lambda$
4: $\mathbf{MASK}_f[id_f] \leftarrow \mathsf{mask}$
5: $\mathsf{value} \leftarrow E_{K_s}(\mathsf{ind}\|w)\|\mathsf{key}_f\|id_f$
6: $b \leftarrow (id_f^*, \mathsf{value} \oplus \mathsf{mask})$
7: $\mathbf{M}_f[\mathsf{ind}].id \leftarrow id_f^*, \mathbf{M}_f[\mathsf{ind}].\mathsf{key} \leftarrow \mathsf{key}$
8: $(id_b, \mathsf{key}_b) \leftarrow (\mathbf{M}_b^*[w_i].id, \mathbf{M}_b^*[w_i].\mathsf{key})$
9: $id_b^* \xleftarrow{\$} \{0,1\}^\lambda$
10: $\mathsf{key}^* \xleftarrow{\$} \{0,1\}^\lambda$
11: $\mathsf{mask} \leftarrow H_2(\mathsf{key}^*, id_b^*)$
12: $b^* \leftarrow (id_b^*, (id_f^*, \mathsf{key}_b, id_b) \oplus \mathsf{mask})$
13: $\mathbf{M}_b^*[w_i].id \leftarrow id_b^*$, $\mathbf{M}_b^*[w_i].\mathsf{key} \leftarrow \mathsf{key}^*$, $\mathbf{M}_b^*[w_i].\mathsf{cnt}$++
14: Send block b, b^* and B_{tail} (*if exists*) to the server.

Server:
15: $\mathbb{D}[b.id] = b.\mathsf{value}, \mathbb{D}[b^*.id] = b^*.\mathsf{value}$
16: **IF**(B_{tail} exists)
17: $\quad \mathbb{D}[B_{\mathsf{tail}}.id] = B_{\mathsf{tail}}.\mathsf{value}$

Update(**delete**, ind, σ, EDB)
Client:
1: $(id, \mathsf{key}) \leftarrow (\mathbf{M}_f[\mathsf{ind}].id, \mathbf{M}_f[\mathsf{ind}].\mathsf{key})$
2: Send (id, key) to the server.

Server:
3: **REPEAT**
4: $\quad b \leftarrow \mathbb{D}[id]$
5: $\quad$ Delete the block b from the dictionary $\mathbb{D}$
6: $\quad \mathsf{mask} \leftarrow \mathbf{MASK}_f(id)$
7: $\quad b.\mathsf{value} \leftarrow b.\mathsf{value} \oplus \mathsf{mask}$
8: $\quad (id, \mathsf{key}) \leftarrow (b.id_b, b.\mathsf{key})$
9: **UNTIL** $(id = \perp)$

Search$(w, i, \sigma, \mathsf{EDB})$ Client:
1: $w_i \leftarrow H(\mathsf{key}_h, w\|i)$
1: $(id, \mathsf{key}) \leftarrow (\mathbf{M}_b^*[w_i].id, \mathbf{M}_b^*[w_i].\mathsf{key})$
2: Send token (id, key) to the server.

Server:
3: $\mathbf{S} \leftarrow$ empty set, $j \leftarrow 0$
4: **REPEAT**
5: $\quad b \leftarrow \mathbb{D}[id]$
6: $\quad \mathsf{mask} \leftarrow H_2(\mathsf{key}, id)$
7: $\quad b.\mathsf{value} \leftarrow b.\mathsf{value} \oplus \mathsf{mask}$
8: $\quad \mathbf{S} = \mathbf{S} \cup b.id_f$
9: $\quad (id, \mathsf{key}) \leftarrow (b.id_b, b.\mathsf{key})$
10: **UNTIL** $(id = \perp)$
11: Send $\mathbf{S}$ to the Client.

Client:
12: $\mathbf{S} \leftarrow \mathsf{Decrypt}_{K_s}(\mathbf{S})$

Fig. 4: Algorithms for G_1.

The computation complexities in Khons are $O(n_w)$ and $O(1)$ in Search and Update process respectively, where n_w is the size of the search result set matching keyword w. And the communication complexities in Khons are also $O(n_w)$ and $O(1)$ in Search and Update process respectively. Khons achieves backward privacy with two round trips.

In Search operation, each partition which has been queried will be updated so that there is no sub-keyword will be repeatedly queried. Except that, the number of elements in each partition is the same, which prevents the server from identifying sub-keyword by the number of elements contained in the partition.

Adaptive security. Khons is the first forward search privacy SSE scheme with partial pattern when . It can also achieve weak backward privacy as the server learns when the deletions occurred. The adaptive security of Khons is proven in Theorem 1.

Theorem 1. Let λ denotes the security parameter. Assume $\mathcal{L}'$ is stateless. Define $\mathcal{L}_{\texttt{Khon}} = (\mathcal{L}_{\texttt{Khon}}^{\mathsf{Search}}, \mathcal{L}_{\texttt{Khon}}^{\mathsf{Update}})$, where

$$\mathcal{L}_{\texttt{Khon}}^{\mathsf{Update}}(\mathsf{op}, w, \mathsf{ind}) = \mathcal{L}'(\mathsf{op}, w),$$

$$\mathcal{L}_{\texttt{Khon}}^{\mathsf{Search}}(w) = \mathcal{L}''(\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{Updates}(w)),$$

Then Khons is $\mathcal{L}_K$-*adaptively-secure* with forward update privacy and backward privacy(BP-II).

Proof 1. We derive some games from real world game to prove the theorem.

Game G_0. G_0 is the real world SSE security game SSEREAL. That is to say,

$$\mathsf{Pr}[\mathrm{SSEREAL}_{\mathcal{A}}^{\texttt{Khons}}(\lambda) = 1] = \mathsf{Pr}[G_0 = 1].$$

Game G_1. In G_1, we pick random strings as identifiers in Update protocol instead of calling H_1 to generate a new encryption key. The algorithm of G_1 is described in Fig. 4. In Search protocol, the random oracle H_2 is programmed so that $H_2(\mathsf{key}, id) = \mathsf{mask}$. Note that in G_1, the generation of key is as same as in G_0, so that mask can be treated as a random string. For convenience, we ignore the generation and application of tail block in G_1. We remove the code which is useless with the security analysis. Furthermore, compared with G_0, we do not consider the reuse of blocks and the partition mechanism in G_1. Hence, we have

$$\mathsf{Pr}[G_0 = 1] - \mathsf{Pr}[G_1 = 1] = 0.$$

Game G_2. In G_2, the same argument of H_2 can be reused. Thus the only difference between G_1 and G_2 is H_2. Hence, we have

$$\mathsf{Pr}[G_1 = 1] - \mathsf{Pr}[G_2 = 1] = 0.$$

Game G_3. In G_3, the same argument of H_3 can be reused. Thus the only difference between G_2 and G_3 is H. Hence, we have

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2894411, IEEE Transactions on Dependable and Secure Computing

Update$(\mathsf{add}, \mathsf{ind}, |W|)$
Client:
1: **For** $(u$ to $u+|W|)$
2: $\mathsf{Update[ind]} \leftarrow u$
3: $\mathsf{Bfid}[u] \xleftarrow{\$} \{0,1\}^{\lambda}$
4: $\mathsf{Bfkey}[u] \xleftarrow{\$} \{0,1\}^{\lambda}$
5: $\mathsf{Bfdata}[u] \xleftarrow{\$} \{0,1\}^{(2\lambda+l)}$
6: $\mathsf{Bf}[u] \leftarrow (\mathsf{Bfid}, \mathsf{Bfdata})$
7: **End For**
8: **For**$(u+|W|$ *to* $u)$
9: Program H_1 *s.t.* $H_1(\mathsf{Bfid}[u]||\mathsf{Bfkey}[u]) \leftarrow \mathsf{Bfdata}[u-1] \oplus (e||\mathsf{Bfid}[u-1]||\mathsf{Bfkey}[u-1])$
10: **For** $(u$ *to* $u+|W|)$
11: $\mathsf{Bwid}[u] \xleftarrow{\$} \{0,1\}^{\lambda}$
12: $\mathsf{Bwkey}[u] \xleftarrow{\$} \{0,1\}^{\lambda}$
13: $\mathsf{Bwkey}^{\#}[u] \xleftarrow{\$} \{0,1\}^{\lambda}$
14: $\mathsf{Bwdata}[u] \xleftarrow{\$} \{0,1\}^{(4\lambda+l)}$
15: $\mathsf{Bw}[u] \leftarrow (\mathsf{Bwid}, \mathsf{Bwdata})$
16: **End For**
17: Send block Bf,Bw to server

Setup():
1: $K_s \xleftarrow{\$} \{0,1\}^{\lambda}$
2: $\mathbf{M}_b$, $\mathbf{M}_b^*$, $\mathbf{M}_f \leftarrow$ empty map
3: $\mathbb{T} \leftarrow$ empty tree

Update(**delete**, ind, $|W|$)
Client:
1: $u \leftarrow \mathsf{Update[ind]}$
2: token $\leftarrow (u, \mathsf{Bfid}[u], \mathsf{Bfkey}[u])$
3: Send token to server.

Search$(\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{Updates}(w))$
Client:
1: $\overline{w} \leftarrow \mathsf{min}(\mathsf{sp}(w))$
2: $(u_0, u_1, \ldots, u_{\mathsf{head}}) \leftarrow \mathsf{Updates}(\overline{w})$
3: **FOR** $(u_i, \mathsf{ind}) \in \mathsf{TimeDB}(w)$ **DO**
4: $e \leftarrow E_{\mathsf{Bwkey}^{\#}[u_i]}(\mathsf{ind})$
5: Program H_2 *s.t.* $H_2(\mathsf{Bwid}[u_i]||\mathsf{Bwkey}[u_i]) \leftarrow \mathsf{Bwdata}[u_i-1] \oplus (e||\mathsf{Bwkey}[u_{i-1}]||\mathsf{Bwid}[u_{i-1}])$
6: **END FOR**
7: Send token$(\mathsf{Bwid}[u_{\mathsf{head}}], \mathsf{Bwkey}[u_{\mathsf{head}}])$ to server.

Fig. 5: Algorithms for Simulator S.

$$\mathsf{Pr}[G_2 = 1] - \mathsf{Pr}[G_3 = 1] = 0.$$

Simulator. The algorithm of simulator is shown in Fig. 5 and the leakage function is $\mathcal{L}_{\texttt{Khons}}$. We generate a new block whose identifier is picked randomly when performing update operation. When performing Khons.**Update**(add), we use the random oracle and store the relationship between timestamp u and document identifier in table Update. Therefore, when perform Khons.**Update**(delete), simulator can get the document identifier through Update. Hence, we have

$$\mathsf{Pr}[G_3 = 1] - \mathsf{Pr}[\mathrm{SSEIDEAL}^{\texttt{Khons}}_{A,S,L_{\texttt{Khons}}}(\lambda) = 1] = 0.$$

Conclusion.By combining all the contributions from all the games, there exists an adversary $\mathcal{A}$ such that

$$|\mathsf{Pr}[\mathrm{SSEREAL}^{\texttt{Khons}}_{\mathcal{A}}(\lambda)=1] - \mathsf{Pr}[\mathrm{SSEIDEAL}^{\texttt{Khons}}_{\mathcal{A},\mathcal{S},L_{\texttt{Khons}}}(\lambda)=1]| \leq \mathsf{Adv}^{\mathsf{prf}}_{F,\mathcal{A}}(\lambda).$$

We conclude that the probability of result is negl(λ) by assuming that PRF is secure.

6 KHONS-F: SSE SCHEME SUPPORTING FULL QUERY

To support full query, we propose a forward security SSE scheme named Khons-f. It satisfies backward privacy(BP-II). The **Setup** and **Update** operation in Khons-f is almost as the same as Khons, so that we will not repeat these algorithms. The only difference between Khons-f and Khons is **Search** operation, which is shown in Algorithm 5.

In Khons-f, we leverage tail blocks to link all elements in $\mathbb{L}_{w_i}$. As mentioned before, the tail block in $\mathbb{L}_{w_i}$ stores the id and key of head block in $\mathbb{L}_{w_i}$ and key of the tail block of $\mathbb{L}_{w_i-1}$. To perform full query, the client issues token=($\mathbf{M}_b^*[w_{\mathsf{num}_p}].id$, $\mathbf{M}_b^*[w_{\mathsf{num}_p}].\mathsf{key}$, $\mathbf{M}_b^*[w].\mathsf{key}_w$). The server can trace all $\mathbb{L}_{w_i}$ in order of $\mathbb{L}_{\mathsf{num}_p}$ to $\mathbb{L}_0$. Firstly, the server retrieves the blocks in $\mathbb{L}_{\mathsf{num}_p}$ one after another. Secondly, the tail block of $\mathbb{L}_{\mathsf{num}_p}$ can be retrieved to get the pointer information of the head block and the encryption key of the tail block in $\mathbb{L}_{\mathsf{num}_{p-1}}$. Therefore, the server can continue to retrieve all the blocks in $\mathbb{L}_{\mathsf{num}_{p-1}}$. And so on in a similar fashion, all the elements belong to keyword w can be sent to the client. The security analysis of Khons-f is similar to Khons, so that we will not repeat the analysis.

Algorithm 5 Khons.**Search**(w, σ; EDB)

Client:
1: $(\mathsf{key}_w, \mathsf{num}_p) \leftarrow (\mathbf{M}_b[w].\mathsf{key}, \mathbf{M}_b[w].\mathsf{num}_p)$
2: $w_i \leftarrow H(\mathsf{key}_h, w||\mathsf{num}_p)$
3: $(id, \mathsf{key}) \leftarrow (\mathbf{M}_b^*[w_i].id, \mathbf{M}_b^*[w_i].\mathsf{key})$
4: $\mathbf{M}_b^*[w_i].\mathsf{flag} \leftarrow$ true
5: Send token $(id, \mathsf{key}, \mathsf{key}_w)$ to the server.

Server:
6: $\mathbf{S} \leftarrow$ empty set, $j \leftarrow 0$
7: **REPEAT**
8: $b \leftarrow \mathbb{D}[id]$
9: $\mathsf{mask}_2 \leftarrow H_2(\mathsf{key}, id)$
10: $b.\mathsf{value} \leftarrow b.\mathsf{value} \oplus \mathsf{mask}_2$
11: $\mathbf{S} = \mathbf{S} \cup \mathbb{D}[b.id_f]$
12: $(id, \mathsf{key}) \leftarrow (b.id_b, b.\mathsf{key})$
13: **IF** $(\mathsf{key} = \perp)$
14: **IF** $(\mathsf{key}_w = \perp)$ $id \leftarrow \perp$
15: **ELSE** $b^* \leftarrow \mathbb{D}[id]$
16: $\mathsf{mask}_3 \leftarrow H_3(\mathsf{key}_w, w)$
17: $b^*.\mathsf{value} \leftarrow b^*.\mathsf{value} \oplus \mathsf{mask}_3$
18: $(id, \mathsf{key}, \mathsf{key}_w) \leftarrow (b^*.id, b^*.\mathsf{key}, b^*.\mathsf{key}_w)$
19: **UNTIL** $(id = \perp)$
20: Send **S** to the Client.

Client:
21: $\mathbf{S} \leftarrow \mathsf{Decrypt}_{K_s}(\mathbf{S})$

7 APPLICATIONS

As a special type of SSE, Khons can be applied to typical ciphertext retrieval scenarios to reduce information leakage or improve the efficiency.

Build secure encrypted applications. The two most popular applications supporting keyword search are mail system and cloud storage system. In encrypted mail system (such as ShadowCrypt [8]), mails are stored in chronological order and queries are allowed to be made within a certain period of time. Its default query is to first get all of the page information and mailing list matching this query on the first page. Then, it obtains the mailing list according to the page selected by the user. Each page can be regarded as a logical partition of Khons. As for cloud storage systems, most of them support pagination display at their clients. Therefore, Khons can also be used to build index trees for documents added in different time periods or different storage areas.

These applications usually do not need to retrieve all documents and pagination query can meet the application requirements. For them, Khons can be directly applied to achieve forward search privacy without loss of functionality. In this case, they can resist some statistical attacks, such as non-adaptive file injection attack.

New directions for designing encrypted databases. Recently, encrypted databases [1, 3–6] have become a promising direction, which provides confidentiality and functionality by running queries on encrypted data based on SSE, order-preserving encryption (OPE) and other cryptographic prototypes. Working as proxy services, they usually build their own indexes for ciphertext data and achieve transparent client support by reinterpreting SQL statements.

In encrypted databases, Khons may help implement some complex queries. Especially, Khons can be considered for implementing pagination query. By reinterpreting SQL "LIMIT" statement to that in one or many partitions, it can reduce information leakage and improve search efficiency. Even if full query must always be executed, Khons can also be used instead of SSE to improve efficiency because it can provide the ability of parallel query. Moreover, in the commercial database products, Khons can be combined with some fine-grained access control techniques like attribute-based encryption (ABE) to enhance the security without loss of efficiency.

8 EXPERIMENTS AND EVALUATION

8.1 Experiment Details

Implementation details. We implemented Fides [24] and Dual [25] in single thread mode without remote RPC. We further implemented Khons in parallel mode (on search operation). Fides [24] is implemented following the Algorithm 2 mentioned in [24] which is based on the open source code of Sophos. Dual [25] is implemented following its pseudo code and is added two-roundtrip mechanism to achieve backward privacy.

For client storage, we used a C++ STL map to store keyword dictionary and a memory-mapped disk-resident hash table to store **M**. For server storage, we chose google implemented B-tree map [31], MongoDB and RocksDB. We wrote wrapper code for them and tested the performance of the above SSE schemes on them.

For cryptographic algorithms, AES and Blake2b are selected as symmetric encryption algorithm and underlying hash function, respectively. The encryption key of AES is set to 256 bits. Moreover, the *sse* crypto library used in Sophos [20] is used as our cryptographic tools.

Experiment environment. All the experiments were performed on a desktop computer with a single Intel Core i7-7700 3.6GHz CPU, 16GB of DDR3 RAM running Linux Ubuntu 14.04 LTS operating system.

8.2 Dataset

We used the well-known Enron email [32] and Wikipedia dumps [33] as our test datasets. TABLE 2 shows their statistic characteristics.

TABLE 2: Dataset overall

	Small dataset	Large dataset
name	$Enron\ email$	$wikipedia$-20150602
tar.gz file size	0.432GB	11.9GB
key-value pair number	34510k	445505k
file number	517k	5078k
key number	20k	70k

Dataset preprocess. A set of keyword-document pairs were extracted from each wikipedia document or email. We used the NLTK library to exclude stopwords and punctuation marks from the original text. Then we used PorterStemmer provided by the NLTK to extract keywords and exclude duplicate keywords in every document.

8.3 Evaluation on B-tree Map

We focus on the detailed tree operations in different SSE schemes, for understanding their effects on overall performance. Meanwhile, we try to evaluate the performance of SSE schemes excluding disk I/O latency. Thus, we used B-tree map [31] stored in memory as server storage structure.

8.3.1 EDB Creation

TABLE 3: Comparison with creation using Enron dataset

Implementation	Time(s)	Pairs per sec(s)	storage(MB)	
			Client	Server
Khons	406	85000	11	3418
Fides	39653	870	16	803
Dual	469	73582	11	2352

We ran all three SSE schemes to store the entire encrypted contents of Enron email dataset. TABLE 3 presents that Khons is 100× faster than Fide [24] and 1.15× faster than Dual [24]. Notice that the time of creation is nearly proportionate to the cryptographic computation time. Compared to Fides [24], AES encryption adopted in Khons is much faster than its 2048-bit RSA-based operation. Compared to Dual [25], Khons does less symmetric cryptographic computation reported in TABLE 4. We also report the fact that our scheme takes up more space than other two schemes.

8.3.2 EDB Search

To evaluate search performance, we searched all keywords extracted from Enron dataset. We further added some code to google B-tree implementation to record the numbers of node splitting, node merging and node rebalancing per

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2894411, IEEE Transactions on Dependable and Secure Computing

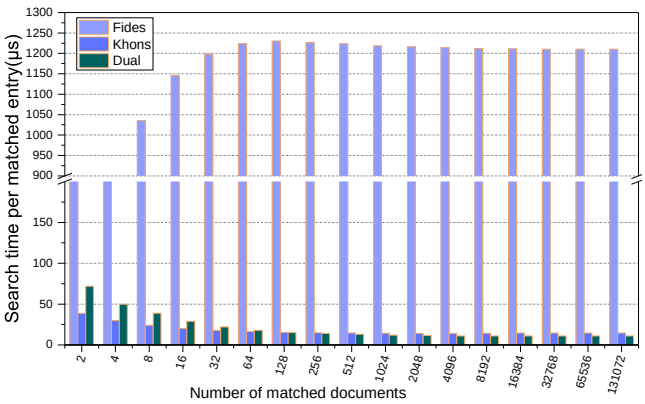


Fig. 6: Comparison with search time per matched document on B-tree

search operation. Due to space constraints, we just reported the records that were not approximate to zero in Fig. 7.

Fig. 6 shows the average time used to search based on the number of documents returned in a search. The average time means the time taken to search divided by the number of matched documents. From Fig. 6, we can see that: 1) `Khons` is 20-100× faster than Fides [24] for all the cases of matched documents; 2) `Khons` is 2× faster than Dual [25] for the case of small number of matched documents, but is nearly equal to Dual [25] for the case of large number of matched documents. We pointed out that Fides [24] and Dual [25] need to delete old nodes and add new nodes. The former is for actual deletion and the latter is in nature. We explain the above conclusion from two aspects. First, the performance of search operation is related to the cryptographic computation. Fides [24] and Dual [25] need to generate indexes for new nodes, but `Khons` can reuse the existing nodes and thus reduce these cryptographic computations. Second, the frequent add and delete operations in Fides [24] and Dual [25] bring more underlying tree operations and corresponding I/O cost. We report the details in TABLE 4. Besides, we could see that the performance of `Khons` is better when considering the real-world I/O latency in section 8.4.

Fig. 7 represents that a significant amount of B-tree nodes is influenced in the search operations of Fides [24] and Dual [25]. `Khons` influences none due to node reuse. The result indicates that node reuse can avoid a significant amounts of storage space allocation, free and memory copy. Thus, node reuse can efficiently reduce the I/O load.

TABLE 4: Comparison of main operations per search

	cryptographic computation				database operation		
	T	H	F	E	insert	delete	update
Fides	$1a_w$	$3a_w$	a_w	a_w	a_w	n_w	0
Dual	-	$5a_w$	-	$1a_w$	$2a_w$	$2a_w$	0
Khons	-	$1a_w$	a_w	$2a_w$	0	0	0

T: trapdoor permutation, H:hash function, F:PRF, E:symmetric encryption

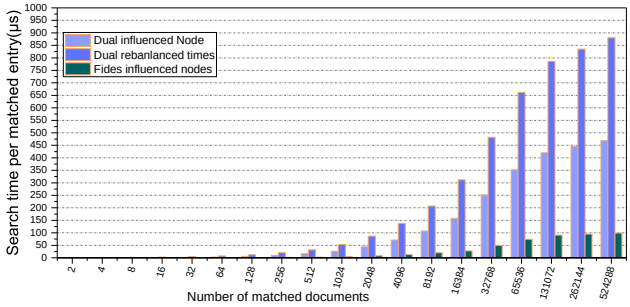


Fig. 7: Comparison with influenced nodes of B-tree. Influenced node means node is allocated or freed; rebalanced node means node whose values inside is moved.

8.4 Evaluation on Real-world Database

To verify that node reuse can take advantage in a real-world database and evaluate the performance in more realistic situation, we stored the data in the well-known key-value database MongoDB, which uses B-tree as its index structure. We also used RocksDB as storage structure to verify whether the node reuse can rise performance in other storage structures, such as LSM tree-based structure.

8.4.1 EDB Creation

We ran all four schemes to store the entire encrypted contents of the Enron email dataset. Fig. 8 shows the EDB creation time of these schemes on different databases.

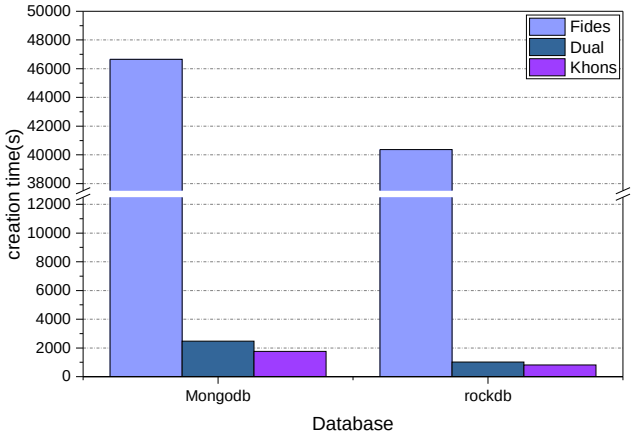


Fig. 8: Comparison with creation using Enron email dataset on real-world database

From Fig. 8, we can see that `Khons` are 20-40× faster than Fides [24] and 1.2× faster than Dual [25]. Compared to the result on B-tree, the performance reduction is due to the database management expenditure and the load of disk I/O. Moreover, the creation time in RocksDB is shorter than that in MongoDB, because the underlying structure of RocksDB has been optimized for write operation. For all the storage wrapper code, we used the default configuration.

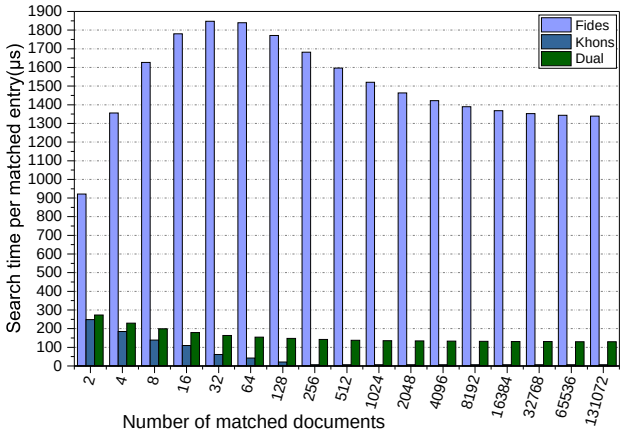


Fig. 9: Comparison with search time on MongoDB (Small Dataset)

8.4.2 EDB Search

From Fig. 9 and Fig. 10, we can see that the disk I/O latency delay factor is enlarged. Compared with Dual [25], Khons is 2-3× faster for all the cases of matched documents. They both much faster than Fides [24].

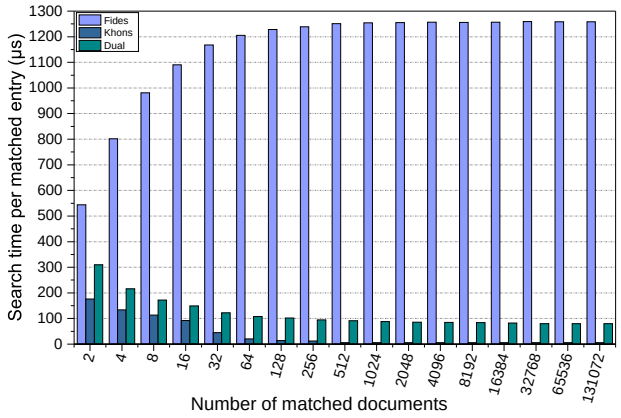


Fig. 10: Comparison with search time on RocksDB (Small Dataset)

8.5 Evaluation on Large Dataset

Experiments on large dataset are designed to evaluate the full search performance of Khons in parallel mode. In these experiments, we chose RocksDB as storage structure and set the maximum number of documents in a partition to 20, which is reasonable in applications supporting pagination query.

8.5.1 EDB Creation

The average throughput of each scheme is close to the performance testing on small datasets. Through experiments, we report that the update throughput of Khons is around 83500 keyword-document pairs per second in average. Dual is around 73000 keyword-document pairs per second and Fides is around 920 keyword-document pairs per second.

8.5.2 EDB Search

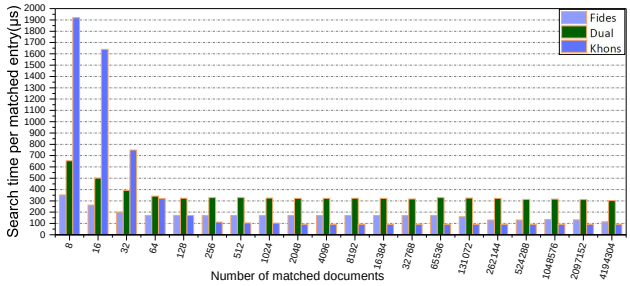


Fig. 11: Comparison with search time per matched document on RocksDB (Large Dataset)

From Fig. 11, we can conclude that Khons is at least 3× faster than Dual [25] and 2× faster than Fides [24] for the cases of medium and large result set. For the case of small result set, however, it is slower than Dual [25] and Fides [24].

In these experiments, we mainly take two factors into considerations: storage accesses and cryptographic computation. For the first, because it is impossible to load full EDB in the case of the large dataset, accessing data on different hierarchies of memory is unavoidable and becomes a bottleneck. Furthermore, since SSE is not optimized for the memory locality, the time spent on memory access is proportional to the number of database operations. Therefore, Dual costs the most time on memory access to complete its search and Khons costs the least. To note, Khons needs more storage space to store node states which leads to extra time cost due to swap of memory. For the second, the parallelized cryptographic computation cost takes up the most of computation cost. For Fides, although RSA and hash function computations are fully parallelized, computation based on RSA private key is very expensive. For Khons, its parallelization mechanism is based on pagination technique. In the case of small result set, the search time cost cannot be amortized. Thus, Fides costs the most time on computation to complete its search while Dual and Khons cost less.

9 Conclusions

In this paper, we proposed the notion "forward search privacy", which ensures search operation over newly added documents doesn't leak the past query information. To achieve this security goal, we developed the new forward private technique, hidden pointer technique (HPT). Finally, we constructed the Khons scheme achieving both forward search privacy and backward privacy. Experiment results show that Khons is efficient and practical.

Khons can support partial query, but the application scenarios of our solutions may be relatively limited. And our scheme can only achieve weak forward search security. How to achieve strong forward search security can be our key point in the future work.

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation Projects (No. 61472091, No. 61672300), for Outstanding Youth Foundation (No. 61722203), Guangzhou scholars project for universities of Guangzhou (No. 1201561613) and National Natural Science Foundation of Tianjin (No. 16JCYBJC15500).

REFERENCES

[1] CipherCloud, "Cloud data encryption", URL: *http://www.ciphercloud.com/technologies/encryption/*.

[2] M. Bellare, A. Boldyreva, A. ONeill. Deterministic and Efficiently Searchable Encryption. In *CRYPTO*, pages 535-552, 2007.

[3] S. Tu, M. F. Kaashoek, S. Madden and N. Zeldovich. Processing analytical queries over encrypted data. In *VLDB*, pages 6(5): 289-300, 2013.

[4] R. A. Popa, C. Redfield, N. Zeldovich and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85-100, 2011.

[5] S. Faber, S. Jarecki, H. Krawczyk, N. Quan, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *ESORICS*, pages 123-145, 2015.

[6] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. C. Rosu and M. Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*, pages 23-26, 2014.

[7] I. Demertzis, D. Papadopoulos, C. Papamanthou. Searchable Encryption with Optimal Locality: Achieving Sublogarithmic Read Efficiency. In *CRYPTO*, pages 371-406, 2018.

[8] W. He, D. Akhawe, S. Jain, E. Shi and D. Song. Shadowcrypt: Encrypted web applications for everyone. In *CCS*, pages 1028-1039, 2014.

[9] R. Li and A. X. Liu. Adaptively secure conjunctive query processing over encrypted data for cloud computing. In *ICDE*, pages 697-708, 2017.

[10] S. Garg, P. Mohassel and C. Papamanthou. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In *CRYPTO* pages 563-592, 2016.

[11] D. X. Song, D. Wagner and A. Perrig. Practical techniques for searches on encrypted data. In *S&P*, IEEE, pages 44-55, 2000.

[12] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis and M. Garofalakis. Practical private range search revisited. In *SIGMOD*, ACM, pages 185-198, 2016.

[13] E. Stefanov, C. Papamanthou and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, pages 72-75, 2014.

[14] M. S. Islam, M. Kuzu and M. Kantarcioglu. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS*, 2012.

[15] D. Cash, P. Grubbs, J. Perry and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, ACM, pages 668-679, 2015.

[16] Y. Zhang, J. Katz and C. Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX Security*, pages 707-720, 2016.

[17] G. Kellaris, G. Kollios, K. Nissim and A. O'Neill. Generic attacks on secure outsourced databases. In *CCS*, ACM, pages 1329-1340, 2016.

[18] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. V. Dijk and S. Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security*, pages 415-430, 2015.

[19] X. Wang, H. Chan and E. Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *CCS*, ACM, pages 850-861, 2015.

[20] R. Bost. $\Sigma o\phi o\varsigma-$Forward Secure Searchable Encryption. In *CCS*, ACM, pages 1143-1154, 2016.

[21] S. Kamara, C. Papamanthou and T. Roeder. Dynamic searchable symmetric encryption. In *CCS*, ACM, pages 965-976, 2012.

[22] K. Kurosawa and Y. Ohtaki. UC-Secure Searchable Symmetric Encryption. In *Financial Cryptography*, pages 285-298, 2012.

[23] M. Naveed, M. Prabhakaran and C. A. Gunter. Dynamic searchable encryption via blind storage. In *S&P*, IEEE, pages 639-654, 2014.

[24] R. Bost, B. Minaudy and O. Ohrimenko. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *CCS*, ACM, pages 1465-1482, 2017.

[25] K.S. Kim, M. Kim, D. Lee, J.H. Park and W.H. Kim. Forward Secure Dynamic Searchable Symmetric Encryption with Efficient Updates. In *CCS*, ACM, pages 1449-1463, 2017.

[26] O. Goldreich, S. Goldwasser and S. Micali. How to construct random functions (extended abstract). In *FOCS*, pages 464-479, 1984.

[27] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov and Y. Huang. Oblivious data structures. In *CCS*, ACM, pages 215-226, 2014.

[28] D. S. Roche, A. Aviv and S. G. Choi. A practical oblivious map data structure with secure deletion and history independence. In *S&P*, IEEE, pages 178-197, 2016.

[29] M. Naveed, S. Kamara and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *CCS*, ACM, pages 644-655, 2015.

[30] J. Katz and Y. Lindell. Introduction to Modern Cryptography. Chapman & Hall/CRC, Boca Raton, FL, 2008.

[31] Google.2011.cpp-btree, https://code.google.com/archive/p/cpp-btree/.

[32] Enron Email Dataset, https://www.cs.cmu.edu/enron.

[33] Wikimedia Foundation, https://dumps.wikimedia.org.

[34] S Navathe. Vertical partitioning algorithms for database design. In *Acm Transactions on Database Systems*, pages 9(4): 680-710, 1984.

[35] C. Curino, E. Jones, Y. Zhang and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, pages 3(1): 48-57, 2010.

[36] M. Stonebraker, A. Aboulnaga, A. Pavlo, A. J. Elamore, R. Taft and M. Serafini. Clay: fine-grained adaptive partitioning for general database schemas. In *VLDB*, pages 10(4): 445-456, 2016.

[37] Y. Lu, A. Shanbhag, A. Jindal and S. Madden. AdaptDB: adaptive partitioning for distributed joins. In *VLDB*, pages 10(5): 589-600, 2017.

[38] Q. Wang, M. He, M. Du, S. Chow, R. Lai and Q. Zou. Searchable Encryption over Feature-Rich Data. In *IEEE Transactions on Dependable and Secure Computing*, pages 15(3): 496-510, 2018.

[39] M. Du, Q. Wang, M. He and J. Weng. Privacy-Preserving Indexing and Query Processing for Secure Dynamic Cloud Storage. In *IEEE Transactions on Information Forensics and Security*, pages 13(9): 2320-2332, 2018.

Zheli Liu received the BSc and MSc degrees in computer science from Jilin University, China, in 2002 and 2005, respectively. He received the PhD degree in computer application from Jilin University in 2009. After a postdoctoral fellowship in Nankai University, he joined the College of Cyber Science of Nankai University in 2011. Currently, he works at Nankai University as a Associate Professor. His current research interests include applied cryptography and data privacy protection.

Jin Li is currently a professor and vice dean of School of Computer Science, Guangzhou University. He received his B.S. (2002) and M.S. (2004) from Southwest University and Sun Yat-sen University, both in Mathematics. He got his Ph.D degree in information security from Sun Yat-sen University at 2007. His research interests include design of secure protocols in Cloud Computing and cryptographic protocols. He has published more than 100 papers in international conferences and journals, including IEEE INFOCOM, IEEE TIFS, IEEE TPDS, IEEE TOC and ESORICS etc. His work has been cited more than 10000 times at Google Scholar and the H-Index is 34. He also served as program chairs and committee for many international conferences. He received NSFC Outstanding Youth Foundation in 2017.

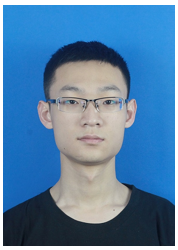
Changyu Dong received the Ph.D. degree from Imperial College London. He is currently a Senior Lecturer with the School of Computing, Newcastle University. He has authored over 30 publications in international journals and conferences. His research interests include applied cryptography, trust management, data privacy, and security policies. His recent work focuses mostly on designing practical secure computation protocols. The application domains include secure cloud computing and privacy preserving data mining.

Yanyu Huang received Bachelor Degree of Information Security from China University of Geosciences, Wuhan, China, in 2016. Currently, she studies for the doctor degree in computer science at Nankai University. Her research interests include applied cryptography, data privacy protection.

Wenjing Lou received her Ph.D. in Electrical and Computer Engineering from the University of Florida. She joined the Electrical and Computer Engineering department at Worcester Polytechnic Institute as an assistant professor in 2003, where she was promoted to associate professor with tenure in 2009. In 2011, she joined the Computer Science department at Virginia Tech as an associate professor with tenure. Her current research interests are in the area of cyber security, with emphases on wireless network security, security and privacy in cloud computing and cyber physical systems. She is also interested in network protocols.

She is currently serving on the editorial board of five journals: IEEE Transactions on Wireless Communications, IEEE Transactions on Smart Grid, IEEE Wireless Communications Letter, Elsevier Computer Networks, and Springer Wireless Networks. She has served as TPC co-chair for the security symposium of several leading IEEE conferences, including General Symposium at IEEE Globecom 2007, Network Security and Privacy Track at IEEE ICCCN 2009, Security Symposium at IEEE ICC 2010, Security and Localization Track at IEEE PIMRC 2011, and Security Symposium at IEEE Globecom 2012. She serves as TPC member regularly for many premier IEEE and ACM conferences.

She was named Joseph Samuel Satin Distinguished fellow in 2006 by WPI. She was a recipient of the U.S. National Science Foundation Faculty Early Career Development (CAREER) award in 2008. She received the Sigma Xi Junior Faculty Research Award at WPI in 2009.



Yu Wei received Bachelor Degree of Information Security and Law from Nankai University, Tianjin, China, in 2018. Currently, he studies for the master degree in computer science at Nankai University. His research interests include applied cryptography, data privacy protection.

Siyi Lv received Bachelor Degree of Information Security and Law from Nankai University, Tianjin, China, in 2016. Currently, she studies for the master degree in computer science at Nankai University. Her research interests include applied cryptography, data privacy protection.