Towards Efficient Verifiable Forward Secure Searchable Symmetric Encryption

Zhongjun Zhang[1,2], Jianfeng Wang[1,2](✉), Yunling Wang[1], Yaping Su[1], and Xiaofeng Chen[1,2]

[1] State Key Laboratory of Integrated Service Networks (ISN), Xidian University, Xi'an 710071, China
zhong_jun@163.com, {jfwang,xfchen}@xidian.edu.cn, ylwang0304@163.com, ypingsu@126.com

[2] State Key Laboratory of Cryptology, P. O. Box 5159, Beijing 100878, China

Abstract. Searchable Symmetric Encryption (SSE) allows a server to perform search directly over encrypted data outsourced by user. Recently, the primitive of forward secure SSE has attracted significant attention due to its favorable property for dynamic data searching. That is, it can prevent the linkability from newly update data to previously searched keyword. However, the server is assumed to be honest-but-curious in the existing work. How to achieve verifiable forward secure SSE in malicious server model remains a challenging problem. In this paper, we propose an efficient verifiable forward secure SSE scheme, which can simultaneously achieve verifiability of search result and forward security property. In particular, we propose a new verifiable data structure based on the primitive of multiset hash functions, which enables efficient verifiable data update by incrementally hash operation. Compared with the state-of-the-art solution, our proposed scheme is superior in search and update efficiency while providing verifiability of search result. Finally, we present a formal security analysis and implement our scheme, which demonstrates that our proposed scheme is equipped with the desired security properties with practical efficiency.

Keywords: Searchable encryption · Forward security · Verifiable search

1 Introduction

Cloud computing enables convenient and on-demand network access to a centralized pool of configurable computing resources, which offers seemly unlimited storage and computation sources. With the rapid development of cloud computing, resource-constraint users prefer to outsource their data to the server. Despite its tremendous benefits, the outsourced data suffers from the security and privacy concerns. According to a recent report, Facebook has leaked over 540 million records including users' IDs, account names, and their activities [15].

One possible solution is to encrypt data with traditional encryption techniques before outsourcing. However, it is intractable to perform search over the encrypted data.

Searchable Symmetric Encryption (SSE) enables the server to search directly over the encrypted data, which has been well studied in both academic and industrial community [5,8,13,21]. More specifically, a user outsources the encrypted documents along with a search index to the cloud server. When the user is interested in a keyword, he generates a search token and submits it to the server. Then the server performs search and returns the matched documents. Recently, the notion of dynamic SSE has aroused wide concern, which can support efficient data update in practical application. However, most of the existing dynamic SSE schemes leak some useful information in data search and update phases. Zhang et al. [27] introduced an effective attack that can fully recover the client's queried keywords by injecting a small number of documents.

To resist the above attack, Bost et al. [2] presented a novel forward secure SSE scheme, where each keyword is attached with a state at local. More specifically, the state is used to update the encrypted index when the document that contains the corresponding keyword is inserted. That is, a new state can be derived from the current state, which can be used to locate the updated document identifier. Note that all the previous states can be derived from the current state. It means that the server can match all the corresponding documents with the given state. According to one-way property of trapdoor permutation, anyone cannot infer the future state from the current state. That implies any adversary cannot obtain the relation between the newly inserted document and the previously searched keyword without the latest state. Thus, the forward security can be achieved. Very recently, Song et al. [22] proposed a more efficient forward secure SSE scheme named FAST based on symmetric primitive. We argue that all the above-mentioned schemes are constructed with the honest-but-curious server model. That is, the server is assumed to perform honestly all the search operations. However, the server may be not fully trust and returns a part of the matched documents due to its selfish behavior. To fight against the malicious server, verifiable SSE has received considerable attention [1,6,20,25,26]. Bost et al. [3] proposed a verifiable forward secure SSE scheme called $\Sigma o\phi o\varsigma$-ϵ based on Verifiable Hash Tables (VHT). However, the VHT must be reconstructed each time an update occurs, which causes low update efficiency and high communication cost. To our best knowledge, how to construct an efficient verifiable SSE scheme with forward security remains a great challenge.

1.1 Our Contribution

In this paper, we further study on verifiable forward secure SSE. Our contributions can be summarized as follows:

- We propose an efficient verifiable forward secure SSE scheme based on the primitive of multiset hash functions, which can achieve verifiability of the search result while maintaining forward security. Our proposed construction

supports efficient verification of search by adopting only symmetric cryptographic primitives.
- Compared with the state-of-the-art solution FAST [22], our proposed scheme can generate a new state in a random fashion instead of the (symmetric) encryption operations in the phase of data updating. Furthermore, it achieves optimized search efficiency without involving the corresponding decryption operations.
- We present the formal security proof of our proposed scheme and provide a thorough implementation of it. The experiment results demonstrate that our construction enjoys a good search and update efficiency.

1.2 Related Work

Song et al. [21] introduced the concept of symmetric searchable encryption (SSE). Subsequently, plenty of research have been exploited in [4,5,8,13,17,23,24]. Goh [13] presented the first index-based SSE scheme to enhance the search efficiency. Curtmola et al. [8] presented the formal security definition of SSE. Cash et al. [5] designed the first sublinear SSE scheme that support boolean queries. Recently, several attacks on SSE have been devised [12,16,27], where the leakage information of data update can be used to recover the client's query. To resist the mentioned attack, several forward secure SSE schemes have been designed to protect the linkability from newly update data to previously searched keyword. Setfanov et al. [23] first formalized the notion of forward security for SSE. Bost [2] presented an efficient forward secure SSE scheme based on the work of [23]. Nevertheless, the computation cost of trapdoor permutation become the primary performance bottleneck. Very recently, Song et al. [22] proposed two more practical forward secure SSE schemes (i.e. FAST and FASTIO) by using only symmetric cryptographic primitives.

Another privacy concern about SSE is the correctness and completeness of search result. Chai et al. proposed the notation of verifiable SSE and constructed the first verifiable SSE scheme based on the character tree [6]. Kurosawa and Ohtaki [18] proposed the first verifiable SSE scheme against the active adversary in 2012 and later extended their scheme to support update operation [19]. Wang et al. [26] proposed a verifiable SSE scheme supporting conjunctive keyword search based on the accumulator. Note that all the existing verifiable SSE can only work in the static SSE setting. In [3], Bost et al. proposed the first verifiable forward secure SSE scheme based on the primitive of Verifiable Hash Tables (VHT). We argue that the Bost's scheme suffers from low data update efficiency and high communication cost due to the reconstruction of VHT. In this work, we will further study how to design a practical forward secure SSE in dynamic SSE application.

1.3 Organization

The rest of this paper is organized as follows. In Sect. 2, we present some preliminaries and the security definitions of our scheme. Section 3 gives the details of our proposed scheme. The formal security and efficiency analysis are given in Sect. 4. The performance evaluation and the conclusion are given in Sects. 4.2 and 6 respectively.

Table 1. Notations and Descriptions

Notations	Descriptions
λ	Security parameter
Σ	The map stored on the client side
T	The map stored on the server side
ind	The identifier of document
l	The length of identifier
op	The update operation taken from the set {add, del}
st_i	The i-th state
DB	The representation of the whole database
DB(w)	All identifiers of documents containing keyword w
$\mathrm{DB}_i(w)$	All identifiers of documents containing w under state st_i
d	The number of keyword-document pairs in DB
H	The representation of hash function
F	The representation of pseudo random function
$\mathcal{H}$	The representation of multiset hash functions
$hash$	The output of multiset hash functions
m	The length of $hash$
R	The representation of search result
$proof$	The representation of proof

2 Preliminaries

In this section, we first describe the notations used in this work, as shown in Table 1. Then we give a brief introduction of the multiset hash functions, which is exploited to construct our scheme. After that, we present a formal security definition.

2.1 Multiset Hash Functions

Clarke et al. [7] introduced an efficient cryptographic primitive named Multiset hash functions, which can map a multiset of arbitrary finite size to a string of fixed length. The difference from the standard hash functions is that the input is a multiset rather than a string. The most attractive property of multiset hash functions is incremental. That is, when a new member is added to the multiset, we can quickly update the result without recalculating over the entire new input.

Given a multiset M, a triple of probabilistic polynomial time algorithms $(\mathcal{H}, +_{\mathcal{H}}, \equiv_{\mathcal{H}})$ is a multiset hash functions if it satisfies the following properties:

- **Compression:** $\mathcal{H}$ is a probabilistic algorithm which can map a multiset to a byte array (a byte array of length m). We call the output of $\mathcal{H}$ as *hash*.
- **Incrementality:** If a new element e is added into M, $\mathcal{H}(M)$ can be incrementally calculated on the basis of the previous *hash* without recalculating. Formally, $\mathcal{H}(M \cup \{e\}) \equiv_{\mathcal{H}} \mathcal{H}(M) +_{\mathcal{H}} \mathcal{H}(\{e\})$.
- **Comparability:** For the same input, $\mathcal{H}$ may output different *hashes* since it is a probabilistic algorithm. Therefore, we need $\equiv_{\mathcal{H}}$ to check whether two *hashes* are equal. Formally, the relation $\mathcal{H}(M) \equiv_{\mathcal{H}} \mathcal{H}(M)$ must hold for all multisets M.

In [7], Clarke et al. introduced four constructions of multiset hash functions, which are MSet-Mu-Hash, MSet-Add-Hash, MSet-VAdd-Hash and MSet-XOR-Hash. Specifically, MSet-Mu-Hash is based on multiplication in a finite field, which results in poor efficiency. MSet-Add-Hash and MSet-VAdd-Hash improve the efficiency by replacing the multiplication with addition and vector addition modular a large integer respectively. All the constructions above are multiset-collision resistant (it is difficult to find two multisets to produce the same *hash*). MSet-XOR-Hash is the most efficient one for only using the XOR operation. Besides, it is set-collision resistant (it is difficult to find a set and a multiset to produce the same *hash*). Considering the efficiency and sufficiency of set-collision resistant property in our scheme, we use the MSet-XOR-Hash construction. we present the definition of the MSet-XOR-Hash as follows:

$$\begin{cases} \mathcal{H}(r, M) = H(0, r) \oplus \bigoplus_{m \in M} H(1, m) \\ \mathcal{H}(r, M \cup \{x\}) \equiv_{\mathcal{H}} \mathcal{H}(r, M) +_{\mathcal{H}} \mathcal{H}(r, \{x\}) \equiv_{\mathcal{H}} \mathcal{H}(r, M) \oplus H(1, x) \\ \mathcal{H}(r, M \setminus \{x\}) \equiv_{\mathcal{H}} \mathcal{H}(r, M) -_{\mathcal{H}} \mathcal{H}(r, \{x\}) \equiv_{\mathcal{H}} \mathcal{H}(r, M) \oplus H(1, x) \end{cases}$$

2.2 Verifiable Dynamic Searchable Symmetric Encryption

Searchable symmetric encryption (SSE) scheme allows a client to encrypt his data before outsourcing them to the server while preserving the ability to search on it. Furthermore, a verifiable dynamic SSE scheme allows a client to update in the outsourced database and to verify the integrity of search result. A verifiable dynamic SSE scheme $\Pi = (\mathbf{Setup}, \mathbf{Search}, \mathbf{Update}, \mathbf{Verify})$ consists of three protocols and one algorithm.

- $(K, \sigma; \mathrm{EDB}) \leftarrow \mathbf{Setup}(\lambda, \mathrm{DB}; \perp)$: In this protocol, the client takes a security parameter λ and a database DB as input and outputs (K, σ), where K is the secret key, σ is the client's state. The server outputs EDB which is the encrypted database stored in the server.

- $(\sigma', \mathrm{R}, \mathit{proof}; \mathrm{EDB}') \leftarrow \mathbf{Search}(K, \sigma, w; \mathrm{EDB})$: This is a protocol between a client with input (K, σ, w) and a server with input EDB. After this protocol, the server returns the matched result R and corresponding *proof* to the client. The client's state σ may be updated to σ' and the encrypted database EDB may be updated to EDB$'$.
- $(\sigma'; \mathrm{EDB}') \leftarrow \mathbf{Update}(K, \sigma, \mathit{ind}, w, \mathit{op}; \mathrm{EDB})$: For the update protocol, the client's input is $(K, \sigma, \mathit{ind}, w, \mathit{op})$, where *ind* is the identifier, w is the keyword and *op* is the operation. Notice that $\mathit{op} = \mathit{add}$ or $\mathit{op} = \mathit{del}$ indicates adding or deleting a keyword-document pair. The server's input is the encrypted database EDB. After this protocol, σ and EDB may be updated.
- $(\mathit{Accept}$ or $\mathit{Reject}) \leftarrow \mathbf{Verify}(K, \sigma, \mathrm{R}, \mathit{proof})$: This algorithm takes $(K, \sigma, \mathrm{R}, \mathit{proof})$ as input, where R and *proof* are returned by the server. The algorithm is used to check whether R is both correct and complete. If yes, the algorithm outputs *Accept*. Otherwise, it outputs *Reject*.

2.3 Security Definitions

All SSE schemes inevitably leak information to the server as the result of communication between the client and the server. Hence, we can give the security definition of our scheme by describing the leakage information. Corresponding to $\Pi = (\mathbf{Setup}, \mathbf{Search}, \mathbf{Update}, \mathbf{Verify})$, the leakage function can be defined as $\mathcal{L} = \{\mathcal{L}_{\mathbf{Setup}}, \mathcal{L}_{\mathbf{Search}}, \mathcal{L}_{\mathbf{Update}}, \mathcal{L}_{\mathbf{Verify}}\}$. Let $\mathcal{A}$ be an adversary and $\mathcal{S}$ be a simulator. We can define the following two probabilistic experiments:

- $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$: $\mathcal{A}$ chooses a security parameter λ, then the experiment runs $\mathbf{Setup}(\lambda, \perp; \perp)$ and returns initialized data structure Σ and T to $\mathcal{A}$. $\mathcal{A}$ adaptively chooses a keyword w and generates a query by running the client phase of search protocol. The experiment answers the query by performing $\mathbf{Search}(k_s, w, \Sigma; \mathrm{T})$ and $\mathbf{Verify}(w, \Sigma, \mathrm{R}, \mathit{proof})$, then gives all client outputs to $\mathcal{A}$. As for update operation, $\mathcal{A}$ generates a query and then the experiment answers the query by running $\mathbf{Update}(k_s, \mathit{ind}, w, \mathit{op}, \Sigma; \mathrm{T})$ and returns all outputs to the adversary $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$ as the output of the experiment.
- $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda)$: $\mathcal{A}$ chooses a security parameter λ. Given the leakage function $\mathcal{L}_{\mathbf{Setup}}(\lambda)$, the simulator $\mathcal{S}$ generates the empty data structures Σ and T by running $\mathcal{S}(\mathcal{L}_{\mathbf{setup}}(\lambda))$ and returns them to $\mathcal{A}$. Then $\mathcal{A}$ adaptively chooses a search or verify query q. The simulator answers the query by performing $\mathcal{S}(\mathcal{L}_{\mathbf{Search}}(q))$ or $\mathcal{S}(\mathcal{L}_{\mathbf{Verify}}(q))$. As for update operation, the simulator answers the update query by running $\mathcal{S}(\mathcal{L}_{\mathbf{Update}}(q))$. Finally, the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$ as the output of the experiment.

We say Π is $\mathcal{L}$-adaptively-secure searchable symmetric encryption scheme if for any probabilistic polynomial time adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that:

$$\left| Pr(\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda) = 1) - Pr(\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda) = 1) \right| \leq \mathbf{negl}(\lambda)$$

2.4 Leakage Functions

The design philosophy of SSE schemes is to allow users to search over encrypted data efficiently while revealing as little information as possible. Similar to [2,22], we will describe the security of our scheme with leakage functions $\mathcal{L}$. We suppose that $\mathcal{L}$ keeps the query history Hist $= (\mathrm{DB_i}, q_i)_{i=0}^{Q}$ which contains all queries issued so far and the snapshots of the database corresponding to q_i. The entries are $q_i = (i, w)$ for a search query on keyword w, or $q_i = (i, ind, w, op)$ for a update query, or $q_i = (i, w, \mathrm{R}, proof)$ for a verify query where R and $proof$ are returned by the server. The integer i is the counter of all queries. The access pattern **ap** is defined as $\mathbf{ap}(\mathrm{Hist}) = (t_1, \ldots, t_Q)$. The entries of $\mathbf{ap}(\mathrm{Hist})$ are $t_i = (i, \mathrm{DB}_i(w_i), proof_i)$ for a search query, or $t_i = (i, op_i, ind_i, proof_i)$ for a update query. The query pattern **qp** is defined as $\mathbf{qp}(w) = \{i \mid q_i$ contains w for each q_i in Hist$\}$.

2.5 Forward Security

Informal speaking, forward security requires that the previous search tokens cannot be exploited to match the new update. That is, the data update operation should reveal no information about the keywords. Similar as [22], the definition of forward secure SSE is formalized as follows:

Definition 1. *(Forward Security). An $\mathcal{L}$-adaptively-secure searchable symmetric encryption scheme is forward secure if for an update query $q_i = (ind_i, w_i, op_i)$, the leakage function $\mathcal{L}_{\mathbf{Update}}(q_i) = (i, op_i, ind_i)$.*

3 Verifiable Forward Secure SSE Scheme

In this section, we firstly present the main idea of the proposed construction. Then, we present the proposed verifiable forward secure SSE scheme in detail.

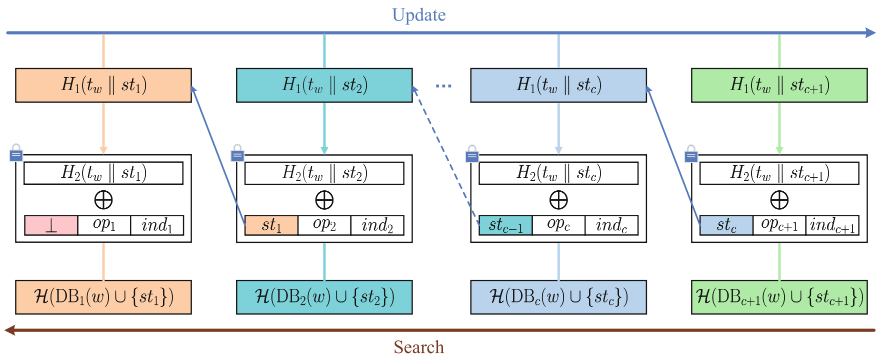

Update
$H_1(t_w \| st_1)$
$H_1(t_w \| st_2)$
$H_1(t_w \| st_c)$
$H_1(t_w \| st_{c+1})$
$H_2(t_w \| st_1)$
$\oplus$
$\perp$ | op_1 | ind_1
$H_2(t_w \| st_2)$
st_1 | op_2 | ind_2
$H_2(t_w \| st_c)$
st_{c-1} | op_c | ind_c
$H_2(t_w \| st_{c+1})$
st_c | op_{c+1} | ind_{c+1}
$\mathcal{H}(\mathrm{DB}_1(w) \cup \{st_1\})$
$\mathcal{H}(\mathrm{DB}_2(w) \cup \{st_2\})$
$\mathcal{H}(\mathrm{DB}_c(w) \cup \{st_c\})$
$\mathcal{H}(\mathrm{DB}_{c+1}(w) \cup \{st_{c+1}\})$
Search

Fig. 1. Update and Search in our Scheme

3.1 Main Idea

In this paper, we propose an efficient verifiable forward secure SSE scheme, which can simultaneously achieve verifiability of search result and forward security property. Our proposed scheme is constructed with the state chain method adopted in $\Sigma o\phi o\varsigma$ [2] and FAST [22]. As shown in Fig. 1, each keyword w corresponds to a state chain, and all identifiers matched w are stored in the locations derived from this chain. When the client wants to search on keyword w, he sends the last state st_{c+1} to the server. Starting from st_{c+1}, the server can get all the previous states $st_c, st_{c-1}, \cdots, st_1$. Hence, the server can walk through the state chain in reverse order and eventually get all the results. Note that the server cannot get the next state st_{c+2} from the current state st_{c+1}, which ensures the forward security property. To further improve the search and update efficiency, the states in our scheme are randomly generated, instead of using permutation functions as in $\Sigma o\phi o\varsigma$ [2] and FAST [22].

3.2 The Concrete Construction

In this section, we present our proposed scheme which mainly consists of three protocols and one algorithm $\Pi = (\textbf{Setup}, \textbf{Search}, \textbf{Update}, \textbf{Verify})$. Suppose that $H_1 : \{0,1\}^* \rightarrow \{0,1\}^n$ and $H_2 : \{0,1\}^* \rightarrow \{0,1\}^{\lambda+l+1}$ are two hash functions, F_1 and F_2 are two pseudo random functions.

- **Setup**$(\lambda, \perp; \perp)$: With the security parameter λ, the client randomly chooses long-term keys k_s and k_r from $\{0,1\}^\lambda$, where k_s is used to generate the search token and k_r serves as the secret key of multiset hash functions. Then the client initializes an empty map Σ as the client storage and the server initializes T as the encrypted database.
- **Search**$(w, \Sigma; \mathrm{T})$: To perform a search query on w, the client firstly retrieves st_c from $\Sigma[w]$, where st_c is the latest state. If st_c is null, this protocol returns an empty set, indicating that no documents matches with w. Otherwise, the client calculates t_w and sends (t_w, st_c) to the server. Upon obtaining (t_w, st_c), the server firstly gets the $proof$ from the last location. Then the server generates all the previous locations and retrieves corresponding search result. Note that the previous states can be obtained directly by decrypting the ciphertext without extra permutation functions like in $\Sigma o\phi o\varsigma$ [2] and FAST [22], which improves the search efficiency.
- **Update**$(\Sigma, ind, w, op; \mathrm{T})$: To add or delete an entry (w, ind), both the client storage Σ and the server storage T need to be updated. For Σ, the client chooses a new state st_{c+1}, and then incrementally generates a new proof $hash'$ on the basis of $hash$. Finally, st_{c+1} and $hash'$ are stored in Σ. For T, the client encrypts the entry to e, and then calculates a new location u from st_{c+1}. Finally, $(u, e, hash')$ is sent to the server to update T.
- **Verify**$(w, \Sigma, \mathrm{R}, proof)$: To verify the integrity of search result R, the client firstly guarantees the returned proof is the latest by comparing it with the one stored in Σ. Besides, the client recomputes multiset hash functions and

Algorithm 1. Verifiable Forward Secure SSE

```
Setup(λ, ⊥; ⊥)
Input: the security parameter $\lambda$
Output: the initialized map $\Sigma$ and T
    Client:
 1: $k_s, k_r \xleftarrow{\$} \{0,1\}^\lambda$, $\Sigma \leftarrow$ empty map
    Server:
 2: $\mathrm{T} \leftarrow$ empty map

Search($k_s, w, \Sigma$; T)
Input: $k_s$, keyword $w$, map $\Sigma$ and T
Output: search result R and the corresponding proof
    Client:
 1: $t_w \leftarrow F_1(k_s, w)$
 2: $(st_c, hash) \leftarrow \Sigma[w]$
 3: if $(st_c, hash) = \perp$
 4:     return $\phi$
 5: end if
 6: send $(t_w, st_c)$ to the server
    Server:
 7: $\mathrm{R}, \Delta \leftarrow \phi$
 8: get the proof from $\mathrm{T}[H_1(t_w \parallel st_c)]$
 9: while $st_c \neq \perp$ do
10:     $u \leftarrow H_1(t_w \parallel st_c)$
11:     $(e, proof_t) \leftarrow \mathrm{T}[u]$
12:     delete $proof_t$ to release storage
13:     $(st_c, ind, op) \leftarrow H_2(t_w \parallel st_c) \oplus e$
14:     if $op = del$ then
15:         $\Delta \leftarrow \Delta \cup \{ind\}$
16:     else if $op = add$ then
17:         if $ind \in \Delta$ then
18:             $\Delta \leftarrow \Delta \setminus \{ind\}$
19:         else
20:             $\mathrm{R} \leftarrow \mathrm{R} \cup \{ind\}$
21:         end if
22:     end if
23: end while
24: send $(\mathrm{R}, proof)$ to the client

Update($k_s, k_r, ind, w, op, \Sigma$; T)
Input: $k_s$, $k_r$, $w$, $ind$, $op$, $\Sigma$, T
Output: possible updated map $\Sigma$ and T
    Client:
 1: $t_w \leftarrow F_1(k_s, w)$, $r_w \leftarrow F_2(k_r, w)$
 2: $(st_c, hash) \leftarrow \Sigma[w]$
 3: if $(st_c, hash) = \perp$ then
 4:     $st_1 \xleftarrow{\$} \{0,1\}^\lambda$
 5:     $e \leftarrow H_2(t_w \parallel st_1) \oplus (\perp \parallel op \parallel ind)$
 6:     $hash' \leftarrow \mathcal{H}(r_w, \{t_w \parallel ind\}) +_{\mathcal{H}} \mathcal{H}(r_w, \{st_1\})$
 7: else
 8:     $st_{c+1} \xleftarrow{\$} \{0,1\}^\lambda$
 9:     $e \leftarrow H_2(t_w \parallel st_{c+1}) \oplus (st_c \parallel op \parallel ind)$
10:     $hash' \leftarrow hash +_{\mathcal{H}} \mathcal{H}(r_w, \{t_w \parallel ind\}) -_{\mathcal{H}} \mathcal{H}(r_w, \{st_c\}) +_{\mathcal{H}} \mathcal{H}(r_w, \{st_{c+1}\})$
11: end if
12: $\Sigma[w] \leftarrow (st_{c+1}, hash')$
13: $u \leftarrow H_1(t_w \parallel st_{c+1})$
14: send $(u, e, hash')$ to the server
    Server:
15: $\mathrm{T}[u] \leftarrow (e, hash')$

Verify($k_r, w, \Sigma, \mathrm{R}, proof$)
Input: $k_r$, $w$, $\Sigma$, R and proof returned from server
Output: Accept or Reject
    Client:
 1: $(st_c, hash) \leftarrow \Sigma[w]$, $r_w \leftarrow F_2(k_r, w)$
 2: $hash' \leftarrow \mathcal{H}(r_w, \{st_c\})$
 3: for all $ind \in \mathrm{R}$ do
 4:     $hash' \leftarrow hash' +_{\mathcal{H}} \mathcal{H}(r_w, \{t_w \parallel ind\})$
 5: end for
 6: if $(proof \equiv_{\mathcal{H}} hash)$ and $(proof \equiv_{\mathcal{H}} hash')$
 7:     return Accept
 8: else
 9:     return Reject
```

compares the output with the returned proof to verify the correctness and completeness of R. If both the two tests above pass, the query integrity can be ensured.

4 Security and Efficiency Analysis

In this section, we first present the security analysis of our work with the simulation-based approach. Then we provide a detailed analysis between our proposed scheme and the state-of-the-art solutions.

4.1 Security Analysis

We introduce the search pattern $\mathbf{sp}(w)$ and update history $\mathbf{uh}(w)$. The search pattern is defined as $\mathbf{sp}(w) = \{i \mid \text{for each}(i, w) \text{ in } Q\}$ which reveals all search queries on w, and the update history $\mathbf{uh}(w) = \{(i, op_i, ind_i) \mid \text{for each}(i, ind_i, w, op_i) \text{ in Hist}\}$. It is clear that $\mathbf{sp}(w)$ and $\mathbf{uh}(w)$ can be obtained from $(\mathbf{ap}(), \mathbf{qp}())$. Differ from [2,22], our scheme extra leaks the proof history $\mathbf{ph}(w) = \{(i, proof_i) \mid \text{for each } (i, ind_i, w_i, op_i) \text{ in Hist}\}$ in the update protocol, which reveals all the proofs corresponding to each update query q_i.

Theorem 1. *Define leakage $\mathcal{L} = \{\mathcal{L}_{\mathbf{Setup}}, \mathcal{L}_{\mathbf{Search}}, \mathcal{L}_{\mathbf{Update}}, \mathcal{L}_{\mathbf{Verify}}\}$ as*

$$\begin{cases} \mathcal{L}_{\mathbf{Setup}} = \perp \\ \mathcal{L}_{\mathbf{Search}} = (\mathbf{sp}(w), \mathbf{uh}(w)) \\ \mathcal{L}_{\mathbf{Update}} = (\mathbf{ph}(w)) \\ \mathcal{L}_{\mathbf{Verify}} = (\mathbf{sp}(w), proof) \end{cases}$$

Then the proposed scheme is a $\mathcal{L}$-adaptively-secure dynamic SSE with forward security.

Proof. The proof of Theorem 1 works by a sequence of indistinguishable hybrids. The first hybrid is the real-world game and the last hybrid is the idea-world game. In the second hybrid, we replace the two PRF F_1 and F_2 by table **Key** and **Rw** respectively. In the third hybrid, H_1 is modeled as a random oracle. We replace the hash function H_1 by randomly chosen strings from $\{0,1\}^n$ in the update protocol. The game will then program the random oracle during the search protocol so that we can produce the right results. Similarly, H_2 is modeled as a random oracle in the fourth hybrid. Note that the multiset hash functions $\mathcal{H}$ is also based on the standard hash function. Hence, the fifth hybrid can model it as a random oracle. The last hybrid only use the predefined leakage functions, which means that we have a simulator which is indistinguishable from the real-world game.

Game G_1: G_1 is the same as $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ except that instead of generating t_w using F, the game maintains a mapping **Key** to store (w, t_w) pairs. Similarly, the game also maintains another mapping **Rw** to store (w, r_w) pairs. In the search protocol, when t_w is needed, the experiment first checks whether there is an entry in **Key** for w. If yes, returns the entry; otherwise randomly picks a t_w in $\{0,1\}^l$ and stores the (w, t_w) pair in **Key**. The operations of **Rw** is the same as **Key**. It's trivial to see that G_1 and $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ are indistinguishable, otherwise we can distinguish a pseudo random function and a truly random function.

$$Pr\left[\mathbf{Real}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda) = 1\right] - Pr(G_1 = 1) \leq \mathrm{Adv}^{\mathrm{prf}}_{F_1,F_2,B_1}(\lambda)$$

Game G_2: Instead of calling hash function H_1 to generate u, G_2 randomly picks a string from $\{0,1\}^n$ and store it in the table **L** in the update protocol.

Algorithm 2. Game G_2

```
Setup($\lambda, \perp; \perp$)
 1: $\Sigma, \mathrm{T} \leftarrow$ empty map
Update($k_s, k_r, ind, w, op, \Sigma; \mathrm{T}$)
     Client:
 2: $t_w \leftarrow \mathbf{Key}[w]$, $r_w \leftarrow \mathbf{Rw}[w]$
 3: $(st_1, \ldots, st_c, hash) \leftarrow \Sigma[w]$
 4: if $(st_1, \ldots, st_c, hash) = \perp$ then
 5:     $st_1 \xleftarrow{\$} \{0,1\}^\lambda$
 6:     $\mathbf{L}[t_w \parallel st_1] \xleftarrow{\$} \{0,1\}^n$
 7:     $e \leftarrow H_2(t_w \parallel st_1) \oplus (\perp \parallel op \parallel ind)$
 8:     $hash' \leftarrow \mathcal{H}(r_w, \{t_w \parallel ind\})$
            $+_{\mathcal{H}} \mathcal{H}(r_w, \{st_1\})$
 9: else
10:     $st_{c+1} \xleftarrow{\$} \{0,1\}^\lambda$
11:     $\mathbf{L}[t_w \parallel st_{c+1}] \xleftarrow{\$} \{0,1\}^n$
12:     $e \leftarrow H_2(t_w \parallel st_{c+1})$
                $\oplus (st_c \parallel op \parallel ind)$
13:     $hash' \leftarrow hash +_{\mathcal{H}} \mathcal{H}(r_w, \{t_w \parallel ind\})$
          $-_{\mathcal{H}} \mathcal{H}(r_w, \{st_c\}) +_{\mathcal{H}} \mathcal{H}(r_w, \{st_{c+1}\})$
14: end if
15: $\Sigma[\mathrm{w}] = (st_1, \ldots, st_c, st_{c+1}, hash')$
16: send $(\mathbf{L}[t_w \parallel st_{c+1}], e, hash')$ to server
     Server:
17: $\mathrm{T}[\mathbf{L}[t_w \parallel st_{c+1}]] = (e, hash')$

Search($k_s, \Sigma, w; T$)
     Client:
18: $t_w \leftarrow \mathbf{Key}[w]$
19: $(st_1, \ldots, st_c, hash) \leftarrow \Sigma[w]$
20: if $(st_1, \ldots, st_c, hash) = \perp$ then
21:     return $\phi$
22: end if
23: for $i = 1$ to $c$ do
24:     $\mathbf{H_1}(t_w \parallel st_i) = \mathbf{L}[t_w \parallel st_i]$
25: end for
26: send $(t_w, st_c)$ to server
     Server:
27: perform search and return $(\mathrm{R}, proof)$
Verify($w, \Sigma, \mathrm{R}, proof$)
     Client:
28: $(st_1, \ldots, st_c, hash) \leftarrow \Sigma[w]$
29: $r_w \leftarrow \mathbf{R}[w]$
30: $hash' = \mathcal{H}(r_w, \{st_c\})$
31: for all $ind \in \mathrm{R}$ do
32:     $hash' = hash' +_{\mathcal{H}} \mathcal{H}(r_w, \{t_w \parallel ind\})$
33: if $(proof \equiv_{\mathcal{H}} hash)$ and $(proof \equiv_{\mathcal{H}} hash')$
34:     return $Accept$
35: else
36:     return $Reject$
37: endif
```

The random oracle is programmed in the search protocol, and we use table $\mathbf{H_1}$ to keep the track of the transcripts. G_2 keeps all st in $\Sigma[w]$ instead of only st_c for the reason that the process of programming $\mathbf{H_1}$ needs the information of all historical state of keyword w. The game is showed in Algorithm 2, note here that we get rid of the server's part in the update protocol.

G_1 and G_2 is indistinguishable except that the query result of random oracle can be inconsistent. Specifically, $\mathbf{L}[t_w \parallel st_{c+1}]$ is generated in the update protocol, but it is lazily programmed to $\mathbf{H_1}$ in the search protocol. Assume that the adversary does not perform any search queries after the update with $t_w \parallel st_{c+1}$, then it queries $\mathbf{H_1}$ with $t_w \parallel st_{c+1}$. Now with an overwhelming probability $\mathbf{L}[t_w \parallel st_{c+1}] \neq \mathbf{H_1}[t_w \parallel st_{c+1}]$. After the next search query, the adversary queries $\mathbf{H_1}$ again with $t_w \parallel st_{c+1}$, now he will get $\mathbf{L}[t_w \parallel st_{c+1}]$. In this case, the inconsistency in the random oracle is observed (we denote this event by **Bad**), hence the adversary knows it is in G_2 instead of the real world game. Thus, we have:

$$\Pr[G_2 = 1] - \Pr[G_1 = 1] \leq \Pr[\mathbf{Bad}]$$

Note that **Bad** happens when the adversary queries the oracle with $t_w \parallel st_{c+1}$. The adversary knows t_w but it has no knowledge about st_{c+1}. The probability for the adversary to choose st_{c+1} is $2^{-\lambda} + negl(\lambda)$. A PPT adversary

Algorithm 3. Simulator S

```
Setup()
1: $v \leftarrow 0$
2: $\mathbf{L}, \mathbf{E}, \mathbf{ST}, \mathbf{P} \leftarrow$ empty map
3: $\mathbf{Key}, \mathbf{Rw}, \Sigma, \mathrm{T} \leftarrow$ empty map
Update($\mathbf{ph}(w)$)
  Client:
4: parse $\mathbf{ph}(w)$ as $[(v_1, proof_1), \ldots, (v_c, proof_c)]$
5: $\mathbf{L}[v] \xleftarrow{\$} \{0,1\}^n$
6: $\mathbf{E}[v] \xleftarrow{\$} \{0,1\}^{\lambda+l+1}$
7: $\mathbf{ST}[v] \xleftarrow{\$} \{0,1\}^{\lambda}$
8: if $v \in [v_1, \ldots, v_c]$
9:     $\mathbf{P}[v] \leftarrow proof_v$
10: else
11:     $\mathbf{P}[v] \xleftarrow{\$} \{0,1\}^m$
12: send $(\mathbf{L}[v], \mathbf{E}[v], \mathbf{P}[v])$ to server
13: $v \leftarrow v + 1$
Verify($\mathbf{sp}(w), proof$)
  Client:
14: $\overline{w} \leftarrow$ min $\mathbf{sp}(w)$
15: $(st_c, hash) \leftarrow \Sigma[\overline{w}]$
16: $t_w \leftarrow \mathbf{Key}[\overline{w}]$, $r_w \leftarrow \mathbf{Rw}[\overline{w}]$
17: $hash' = \mathbf{H_3}(0, r_w) \oplus \mathbf{H_3}(1, st_c)$
18: for all $ind \in \mathrm{R}$ do
19:     $hash' = hash' \oplus \mathbf{H_3}(1, t_w \parallel ind)$
20: if $proof = hash$ and $proof = hash'$
21:     return $Accept$
22: else
23:     return $Reject$
```

```
Search($\mathbf{sp}(w), \mathbf{uh}(w)$)
  Client:
24: $\overline{w} \leftarrow min\ \mathbf{sp}(w)$
25: $t_w \leftarrow \mathbf{Key}[\overline{w}]$
26: $r_w \leftarrow \mathbf{Rw}[\overline{w}]$
27: if $\mathbf{H_3}(0, r_w) = \perp$
28:     $\mathbf{H_3}(0, r_w) \xleftarrow{\$} \{0,1\}^m$
29: end if
30: parse $\mathbf{uh}[w]$ as $[(v_1, op_1, ind_1), \ldots, (v_c, op_c, ind_c)]$
31: $c \leftarrow |\mathbf{uh}[w]|$
32: if $c = 0$
33:     return $\phi$
34: end if
35: $\mathbf{P}[v_0] \leftarrow \mathbf{H_3}(0, r_w)$, $st_0 \leftarrow \perp$
36: $\mathbf{H_3}(1, st_0) \leftarrow 0^m$
37: for $i = 1$ to $c$ do
38:     $st_i \leftarrow \mathbf{ST}[v_i]$, $st_{i-1} \leftarrow \mathbf{ST}[v_{i-1}]$
39:     $\mathbf{H_3}(1, st_i) \xleftarrow{\$} \{0,1\}^m$
40:     program $\mathbf{H_1}$: $\mathbf{H_1}(t_w \parallel st_i) = \mathbf{L}[v_i]$
41:     program $\mathbf{H_2}$: $\mathbf{H_2}(t_w \parallel st_i) = \mathbf{E}[v_i] \oplus (st_{i-1} \parallel op_i \parallel ind_i)$
42:     program $\mathbf{H_3}$: $\mathbf{H_3}(1, t_w \parallel ind_i) = \mathbf{P}[v_i] \oplus \mathbf{H_3}(1, st_i) \oplus \mathbf{P}[v_{i-1}] \oplus \mathbf{H_3}(1, st_{i-1})$
43: end for
44: $\Sigma[\overline{w}] = (\mathbf{ST}[v_c], \mathbf{P}[v_c])$
45: send $(t_w, \mathbf{ST}[v_c])$ to server
  Server:
46: perform search and returns $(\mathrm{R}, proof)$
```

can make at most $\mathbf{poly}(\lambda)$ guesses, then $\Pr[\mathbf{Bad}] \leq \mathbf{poly}(\lambda) \cdot (2^{-\lambda} + negl(\lambda))$. The probability $\Pr[\mathbf{Bad}]$ is negligible so that G_1 and G_2 are indistinguishable.

Game G_3: G_3 does exactly what G_2 did for H_1, but for H_2. We can prove that G_3 and G_2 are perfectly indistinguishable by the same reduction.

$$Pr(G_3 = 1) - Pr(G_2 = 1) \leq negl(\lambda)$$

Game G_4: In G_3, $hash'$ is calculated by multiset hash functions with the knowledge of ind and st_{c+1}. While in G_4, the client randomly chooses a string from $\{0,1\}^m$ as $hash'$ and records it to the table $\mathbf{P}$. G_4 and G_3 behave exactly the same in the adversary's perspective. In the update protocol, both the two games output three uniformly random strings. In the verify algorithm, the client can recalculate the output of $\mathcal{H}(\mathrm{R})$ to get the correct verify result.

$$Pr(G_4 = 1) - Pr(G_3 = 1) = 0$$

Game G_5: In G_5, we introduce a global counter v to record the number of updates operations. v is initialized in the setup protocol and increases in the update protocol. We will show that G_5 and G_4 are indistinguishable. In the update phase, the client generates st, e and $proof$ randomly and sends them to the server in both games. The adversary cannot distinguish the two games for the reason that for every update operation, there are always three new strings sent from the client to server. For the search phase, the client sends two strings st_c and t_w to the server and the server will do the exact same thing.

$$Pr(G_5 = 1) - Pr(G_4 = 1) = 0$$

The Simulator: The simulator, as shown in Algorithm 3, is the same as G_5, except those two places. Firstly, the simulator gets the update history of keyword w from leakage function $\mathbf{uh}(w)$. Secondly, the simulator uniquely maps from keyword w to $\overline{w} = \min \mathbf{sp}(w)$.

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

4.2 Comparison

We compare our scheme with FAST [22], $\Sigma o\phi o\varsigma$ [2], $\Sigma o\phi o\varsigma$-ϵ [2], Janus [4] and Janus++ [24] in this section. All of those schemes can achieve forward security. Janus and Janus++ mainly focus on backward security, but they can achieve forward security as well. Only our scheme and $\Sigma o\phi o\varsigma$-ϵ can ensure verifiability of search result. Our scheme can be viewed as an extension of FAST to improve the search efficiency and to support verifiability of search result.

Table 2. Performance comparison

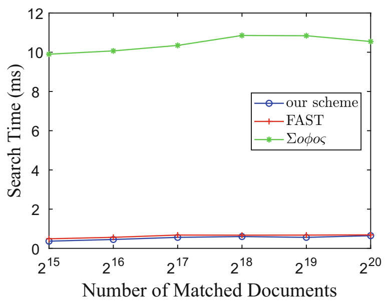
Schemes	FS	BS	Update	Search	Verify
$\Sigma o\phi o\varsigma$[2]	✓	✗	$t_{\mathrm{TP}} + 2t_{\mathrm{H}}$	$\lvert\mathrm{U_w}\rvert \cdot (t_{\mathrm{TP}} + 2t_{\mathrm{H}})$	✗
$\Sigma o\phi o\varsigma$-ϵ[2]	✓	✗	$t_{\mathrm{TP}} + t_{\mathcal{H}} + 2t_{\mathrm{H}}$	$\lvert\mathrm{U_w}\rvert \cdot (t_{\mathrm{TP}} + 2t_{\mathrm{H}})$	$t_{\mathcal{H}} + t_{\mathrm{VHT}}$
FAST [22]	✓	✗	$t_{\mathrm{P}} + 2t_{\mathrm{H}}$	$\lvert\mathrm{U_w}\rvert \cdot (t_{\mathrm{P}} + 2t_{\mathrm{H}})$	✗
Our Scheme	✓	✗	$t_{\mathcal{H}} + 2t_{\mathrm{H}}$	$\lvert\mathrm{U_w}\rvert \cdot (2t_{\mathrm{H}})$	$t_{\mathcal{H}}$
Janus [4]	✓	✓	t_{PPE}	$(\lvert\mathrm{U_w}\rvert - \lvert\mathrm{D_w}\rvert) \cdot t_{\mathrm{PPE}}$	✗
Janus++ [24]	✓	✓	t_{SPE}	$(\lvert\mathrm{U_w}\rvert - \lvert\mathrm{D_w}\rvert) \cdot t_{\mathrm{SPE}}$	✗

We denote by TP a trapdoor permutation, t_{TP} the time cost of TP, P a standard permutation, t_{P} the time cost of P. The public puncturable encryption scheme is denoted as PPE and the time cost of its operations (encrypt, puncture and decrypt included) is denoted as t_{PPE}. The symmetric puncturable encryption scheme is denoted as SPE and the time cost of its operations (encrypt, puncture and decrypt included) is denoted as t_{SPE}. t_{H} stands for the time cost of standard hash function and $t_{\mathcal{H}}$ for the multiset hash functions. t_{VHT} is the time cost of reconstructing the verifiable hash table. $\mathrm{U_w}$ is the number of update operations about keyword w, i.e., the number of times that keyword w was historically added and deleted in the database. D_w is the number of delete operations about keyword w. FS (resp. BS) stands for forward (resp. backward) security. Table 2 gives the performance comparison of the four schemes above.

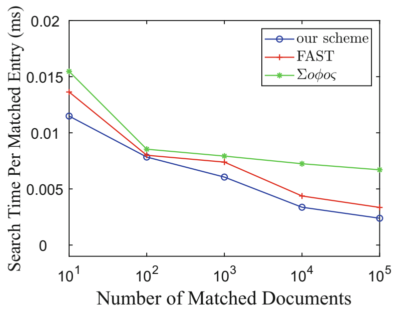
5 Performance Evaluation

In this section, we present a thorough performance evaluation of our proposed scheme. We implement our scheme in C/C++ and use crypto++ [9] library to instantiate the cryptographic operations: AES in CTR model for the PRF F_1 and F_2, SHA256 for the hash function H_1 and H_2. We evaluate the proposed scheme by comparing it with $\Sigma o \phi o \varsigma$ [2], $\Sigma o \phi o \varsigma$-$\epsilon$ [2] and FAST [22]. For all those schemes, we use Rocksdb [10] to store the data in the client and the server, and gRPC [14] to implement the communication between them. The length of symmetric keys are set to 128 bits in all schemes.

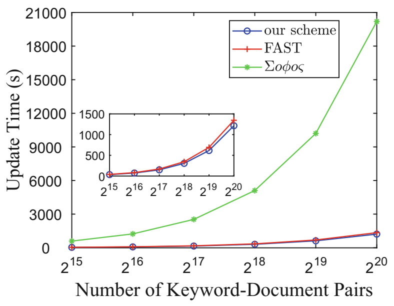
We evaluate our scheme using two LINUX machines, one is service node and the other is client node. Both of them have 4 cores with 8 threads (Intel Xeon E5-1620 v3, 3.50 GHz), 16 GB RAM and 1 TB disk, running on Ubuntu 14.04 LTS. We adopt the open-source C++ implementation of $\Sigma o \phi o \varsigma$ and FAST from GitHub for the comparison. The implementation of our scheme is also an open source project in GitHub [28].



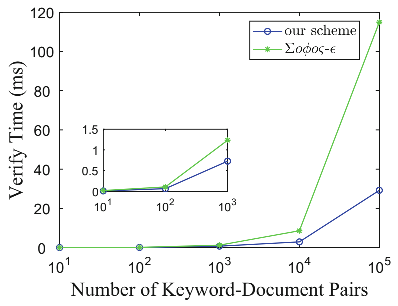
(a) Time cost of Search(real world data).



(b) Time cost of Search (synthetic data).



(c) Time cost of Update.



(d) Time cost of Verify.

Fig. 2. The performance comparison.

Firstly, we consider the real-world dataset from Wikimedia Download [11] to drive our evaluations. The dataset is preprocessed into a Rocksdb file from which the update algorithm reads out keyword-document pairs. There are 786,629 documents and 4,006,619 distinct keywords in the dataset. We perform our evaluation with different database of the size ranging from 2^{15} to 2^{20}. When evaluating the search efficiency, we choose randomly a keyword and search it using three schemes.

To evaluate the efficiency of search and verification in the case a single keyword is included by a large number of documents, we also use 5 synthetic datasets of increasing size. The number of matched documents ranges from 10 to 10^5. When evaluating the search efficiency, we repeat the search process 10 times and take the average, then divided it by the number of matched documents to get the search time per matched entry.

Search Efficiency. Figure 2(a) shows the evaluation results of the search protocol on real-world dataset. As we can see, the search efficiency of $\Sigma o\phi o\varsigma$ is obviously inferior to FAST and our scheme. The reason is that $\Sigma o\phi o\varsigma$ needs to perform a trapdoor permutation, a public key primitive, to get the previous state from the current state. Both our scheme and FAST use only symmetric key operations, but an extra trapdoor permutation operation needs to be performed in FAST. Hence, our scheme performs better than $\Sigma o\phi o\varsigma$ and FAST.

Figure 2(b) shows the evaluation results of the search protocol on synthetic data. As we can see, when the number of matched documents increases, the search time of per matched entry decreases. The reason is that all of those schemes have some initializing operations in search protocol, such as reading out the current state from Σ and generating the search token. Actually, the search complexity of all the three schemes depends only on the number of matched documents. Those one-time costly operations are amortized into every entry in the search result. The search time of per matched entry of $\Sigma o\phi o\varsigma$ is larger than both of our scheme and FAST for the reason mentioned above.

Table 3. Update efficiency

Schemes	$\Sigma o\phi o\varsigma$	FAST	Our scheme
Throughput (ops/s)	3997	43764	78125
Single update time (ms)	0.25	0.023	0.013

Update Efficiency. Figure 2(c) shows the performance of update protocol on real-world dataset. Note that the latencies caused by RPC communication and disk access are included in our evaluations. For example, the update algorithm needs to read out keyword-document from Rocksdb files and the update requests need to be sent to the server. Intuitively, the bottleneck of update protocol is the computation of the new update token. The update time of $\Sigma o\phi o\varsigma$ is the largest for the reason that it needs to perform a trapdoor permutation in the update protocol. For FAST, the update protocol consists of a permutation and two hash functions. For our scheme, the update protocol includes an update operation of multiset hash functions and two hash functions. Note that the update operation of multiset hash functions is some hash function operations in our scheme, which is much more efficient than permutation functions. Hence, our scheme performs better than FAST on update efficiency. The difference of their efficiency has become more and more obvious as the number of keyword-document pairs increases.

To further evaluate the update efficiency, we calculate the throughput of update operation and the update time of single keyword-document pair of the three schemes. As shown in Table 3, the throughput of our scheme is about 1.7 times that of FAST and about 19 times that of $\Sigma o\phi o\varsigma$. The experimental results confirm that the proposed scheme has a better performance in terms of update efficiency.

Verify Efficiency. Figure 2(d) shows the performance of verify operation. Our scheme performs better than $\Sigma o\phi o\varsigma$-ϵ since an extra reconstruction of verifiable hash table is involved in it. As we can see, the difference increases dramatically with the increasing of the number of keyword-document pairs. The reason is that the time cost of reconstructing a hash table is proportional to the number of elements of it.

6 Conclusion

In this paper, we study the problem of forward secure SSE in the malicious setting. A new efficient verifiable forward secure SSE is proposed based on the primitive of multiset hash functions, which can ensure query integrity and forward security. Additionally, we provide a formal security proof to prove that our scheme can achieve the desired security goals. The proposed scheme is implemented and the experiment result shows that it can provide a more efficient performance in search and update processes.

Acknowledgement. This work is supported by National Key Research and Development Program of China (No. 2017YFB0802202), National Natural Science Foundation of China (Nos. 61702401 and 61572382), the Fundamental Research Funds for the Central Universities (XJS17053), National Cryptography Development Fund (No. MMJJ20180110) and China 111 Project (No. B16037).

References

1. Azraoui, M., Elkhiyaoui, K., Önen, M., Molva, R.: Publicly verifiable conjunctive keyword search in outsourced databases. In: Proceedings of 2015 IEEE Conference on Communications and Network Security, CNS 2015, pp. 619–627. IEEE (2015)

2. Bost, R.: $\Sigma o\varphi o\varsigma$: forward secure searchable encryption. In: Proceedings of the 2016 ACM Conference on Computer and Communications Security, CCS 2016, pp. 1143–1154. ACM (2016)
3. Bost, R., Fouque, P., Pointcheval, D.: Verifiable dynamic symmetric searchable encryption: optimality and forward security. IACR Cryptology ePrint Archive 2016, p. 62 (2016). http://eprint.iacr.org/2016/062
4. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, pp. 1465–1482. ACM (2017)
5. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for Boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_20
6. Chai, Q., Gong, G.: Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In: Proceedings of 2012 IEEE International Conference on Communications, ICC 2012, pp. 917–922. IEEE (2012)
7. Clarke, D., Devadas, S., van Dijk, M., Gassend, B., Suh, G.E.: Incremental multiset hash functions and their application to memory integrity checking. In: Laih, C.-S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 188–207. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40061-5_12
8. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. J. Comput. Secur. **19**(5), 895–934 (2011)
9. Dai, W.: Crypto++: A free C++ class library of cryptographic schemes (2019). https://cryptopp.com/. Accessed 10 June 2019
10. Facebook Inc.: Rocksdb: a persistent key-value store for flash and ram storage (2019). http://rocksdb.org Accessed 10 June 2019
11. Foundation, W.: Wikimedia downloads (2019). https://dumps.wikimedia.org. Accessed 10 June 2019
12. Giraud, M., Anzala-Yamajako, A., Bernard, O., Lafourcade, P.: Practical passive leakage-abuse attacks against symmetric searchable encryption. In: Proceedings of the 14th International Joint Conference on e-Business and Telecommunications, pp. 200–211. IEEE (2017)
13. Goh, E.: Secure indexes. IACR Cryptology ePrint Archive 2003, p. 216 (2003). http://eprint.iacr.org/2003/216
14. Google, Inc.: GRPC: a high performance, open-source universal RPC framework (2019). http://www.grpc.io/. Accessed 10 June 2019
15. Hashim, A.: Latest facebook data breach totals over 540 million records found unsecured. https://latesthackingnews.com/2019/04/04/latest-facebook-data-breach-totals-over-540-million-records-found-unsecured/. Accessed 29 Apr 2019
16. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium. NDSS (2012)
17. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012, pp. 965–976. ACM (2012)
18. Kurosawa, K., Ohtaki, Y.: UC-secure searchable symmetric encryption. In: Proceedings of the 16th International Conference on Financial Cryptography and Data Security, FC 2012, pp. 285–298. IEEE (2012)

19. Kurosawa, K., Sasaki, K., Ohta, K., Yoneyama, K.: UC-secure dynamic searchable symmetric encryption scheme. In: Proceedings of the 11th International Workshop on Security Advances in Information and Computer Security, IWSEC 2016, pp. 73–90. IEEE (2016)
20. Ogata, W., Kurosawa, K.: Efficient no-dictionary verifiable searchable symmetric encryption. In: Proceedings of the 21st International Conference on Financial Cryptography and Data Security, FC 2017, pp. 498–516. IEEE (2017)
21. Song, D.X., Wagner, D.A., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of 2000 IEEE Symposium on Security and Privacy, S&P 2000, pp. 44–55. IEEE (2000)
22. Song, X., Dong, C., Yuan, D., Xu, Q., Zhao, M.: Forward private searchable symmetric encryption with optimized I/O efficiency. IEEE Trans. Dependable Secur. Comput. (2018). https://doi.org/10.1109/TDSC.2018.2822294
23. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS (2014)
24. Sun, S., et al.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, pp. 763–780 (2018)
25. Sun, W., Liu, X., Lou, W., Hou, Y.T., Li, H.: Catch you if you lie to me: efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data. In: Proceedings of 2015 IEEE Conference on Computer Communications, INFOCOM 2015, pp. 2110–2118. IEEE (2015)
26. Wang, J., Chen, X., Sun, S.-F., Liu, J.K., Au, M.H., Zhan, Z.-H.: Towards efficient verifiable conjunctive keyword search for large encrypted database. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11099, pp. 83–100. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98989-1_5
27. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: Proceedings of the 25th Security Symposium, USENIX 2016, pp. 707–720. IEEE (2016)
28. Zhang, Z.: Implementation of our scheme (2019). https://github.com/zhangzhongjun/VFSSSE. Accessed 10 June 2019