How to Update Documents *Verifiably* in Searchable Symmetric Encryption

Kaoru Kurosawa and Yasuhiro Ohtaki

Ibaraki University, Japan
{kurosawa,y.ohtaki}@mx.ibaraki.ac.jp

Abstract. In a searchable symmetric encryption (SSE) scheme, a client can store encrypted documents to a server in such way that he can later retrieve the encrypted documents which contain a specific keyword, keeping the keyword and the documents secret. In this paper, we show how to update (modify, delete and add) documents in a *verifiable* way. Namely the client can detect any cheating behavior of malicious servers. We then prove that our scheme is UC-secure in the standard model.

Keywords: keyword search, searchable symmetric encryption, update, verifiable.

1 Introduction

We consider a scheme such as follows [15]: a client stores some files D_i in an encrypted form C_i on a remote server in the store phase. Later, in the search phase, the client can efficiently retrieve the encrypted files containing specific keywords w, keeping the keywords themselves secret and not jeopardizing the security of the remotely stored files. Such a scheme is called a searchable symmetric encryption (SSE) scheme because a symmetric key encryption scheme is used to encrypt files. (For example, a client may want to store old email messages encrypted on a server managed by Google or another large vendor, and later retrieve certain messages while traveling with a mobile device.)

The notion of SSE schemes was introduced by Song et al. [25]. Then after a series of works [25, 17, 1, 15], Curtmola, et al. [10, 11] gave a rigorous definition of privacy against passive adversaries. Namely a server is an advresary who is honest but curious. They then showed two schemes, SSE-1 and SSE2-2, where SSE-1 is more efficient than SSE-2, and SSE-2 is more secure than SSE-1. In particular, SSE-2 is secure against adaptive chosen keyword attacks.

On the other hand, Kurosawa et al. [21] considered a case such that the server is malicious. A malicious server may delete some encrypted files to save her memory space, for example. Even if the server is honest, a virus, worm, trojan horse or a software bug may delete, forge or swap some encrypted files. An adversary would then make a profit if the files are related to bank accounts, tax or some critical information. They [21] then showed a *verifiable* SSE scheme in which the client can detect any cheating behavior of malicious servers.

M. Abdalla, C. Nita-Rotaru, and R. Dahab (Eds.): CANS 2013, LNCS 8257, pp. 309–328, 2013.

In fact, Kurosawa et al. [21] proved that their scheme is UC-secure, where UC (universal composability) is a very strong notion of security. In the UC framework [7–9], the security of a protocol is maintained under a general protocol composition. Therefore their SSE scheme [21] is secure even when it is composed with itself and/or other cryptographic protocols and primitives.

Recently Kamara et al. [23] constructed a *dynamic* SSE scheme such that the client can add and delete documents. They then proved that their scheme is secure against adaptive chosen keyword attacks. Further the search time is sublinear. Subsequently Kamara et al. [22] showed a parallel and dynamic SSE scheme. However, these dynamic schemes [23, 22] are not verifiable. Namely the client cannot detect cheating behavior of malicious servers. (Also the security holds in the random oracle model only.)

In this paper, we first show a more efficient verifiable SSE scheme than Kurosawa et al. [21]. In this scheme, the client sends only $n + 128$ bits in the search phase while $(\log n + \ell + 1) \times n$ bits must be sent in [21], where n is the number of documents and ℓ is the bit length of each keyword.

Table 1. Comparison with The Previous Works

	Curtmola et al. [10]	Kurosawa et al. [21]	Kamara et al. [23, 22]	This paper
Verifiability	×	◯	×	◯
Dynamic (Update)	×	×	◯	◯

We next extend our verifiable SSE scheme to a *verifiable dynamic* SSE scheme. Namely the client can update (modify, delete and add) documents, and he can detect any cheating behavior of malicious servers. See Table 1 for the comparison with the previous works.

We illustrate our idea of the construction by using an example. Suppose that the client wants to search on a keyword *Austin*, and *Austin* is included in three documents D_1, D_3, D_5 whose ciphetexts are C_1, C_3, C_5. In the verifiable SSE scheme of [21], the client sends a query $t(Austin)$ to the server, and the server returns (C_1, C_3, C_5) together with $tag = \mathtt{MAC}(t(Austin), (C_1, C_3, C_5))$, where $t(Austin)$ is some trapdoor information. Namely the client authenticates the whole communication sequence, $t(Austin)$ and (C_1, C_3, C_5). He then stores the authenticator, *tag*, on the server in the store phase.

In this scheme, however, the client cannot modify C_i efficiently. For example, suppose that C_1 includes two keywords, *Austin* and $Washington$. To modify C_1 to C_1', the client must store two updated authenticators, $\mathtt{MAC}(t(Austin), (C_1', C_3, C_5))$ and $\mathtt{MAC}(t(Washington), (C_1', \cdots))$, to the server in the update phase. If C_1 includes more keywords, then the client must updates more authenticators.

Now our idea is that the client authenticates only $(t(Austin), 1, 3, 5)$. He separately authenticates each (i, C_i) also. Then to update C_1 to C_1', the client stores just an authenticator on $(1, C_1')$ to the server. The update cost is only this no matter how many keywords are included in C_1. Thus the client can update each C_i efficiently.

To delete a document C_1, the client updates it to a special symbol $C_1' = delete$ similarly. To add a new document D_6 which includes $Austin$, the client updates the authenticator on $(t(Austin), 1, 3, 5)$ to that on $(t(Austin), 1, 3, 5, 6)$.

Finally, we prove that our verifiable dynamic SSE scheme is UC-secure in the standard model.

1.1 Related Work

Conjunctive keyword search in the SSE setting was first considered by Golle et al. [19]. In their scheme, a client specifies at most one keyword in each keyword field. This framework was followed up by [3, 4]. Wang et al. [26] gave a scheme which does not have such a structure. Recently Cash et al. [12] showed a keyword field free scheme which can support general Boolean queries.

Chase et al. [13] extended and generalized the security model of SSE schemes to complex data (e.g., graphs) and introduced the notion of associated data that allows to compose different components of the protocol.

2 Verifiable Searchable Symmetric Encryption

If X is a string, then $|X|$ denotes the bit length of X. $[X]_{1..u}$ denotes the first u bits of X, and $[X]_u$ denotes the uth bit of X. If X is a set, then $|X|$ denotes the cardinality of X. PPT means probabilistic polynomial time.

2.1 Verifiable SSE Scheme

Let $\mathcal{D} = \{D_1, \cdots, D_n\}$ be a set of documents and $\mathcal{W} = \{w_1, \cdots, w_m\}$ be a set of keywords. Let $\mathtt{Index} = \{e_{i,j}\}$ be an $m \times n$ binary matrix such that

$$e_{i,j} = \begin{cases} 1 \text{ if } w_i \text{ is contained in } D_j \\ 0 \ otherwise \end{cases} . \tag{1}$$

Let $\mathtt{D}(w)$ denote the set of documents which contain a keyword $w \in \mathcal{W}$. Also let $\mathtt{List}(w) = \{i \mid D_i \text{ contains } w\}$.

A verifiable SSE scheme is a protocol between a client and a server as follows.

(Store phase)

On input $(\mathcal{D}, \mathcal{W}, \mathtt{Index})$, the client sends $(\mathcal{C}, \mathcal{I})$ to the server, where $\mathcal{C} = (C_1, \cdots, C_n)$ is the set of encrypted documents, and $\mathcal{I}$ is an encrypted $\mathtt{Index}$.

(Search phase)

1. On input a keyword $w \in \mathcal{W}$, the client sends a trapdoor information $t(w)$ to the server.
2. The server somehow computes $\mathtt{C}(w) = \{C_i \mid D_i \text{ contains } w\}$, and returns $(\mathtt{C}(w), Tag)$ to the client, where Tag is an authenticator.

Real Game ($\mathtt{Game}_{real}$)

- In the store phase, an adversary **A** chooses $(\mathcal{D}, \mathcal{W}, \mathtt{Index})$ and sends them to the challenger. The challenger returns $(\mathcal{I}, \mathcal{C})$.
- In the search phase, for $i = 1, \cdots, q$,
  1. **A** chooses a keyword $w_{a_i} \in \mathcal{W}$ and sends it to the challenger.
  2. The challenger returns a trapdoor information $t(w_{a_i})$ to **A**.
- Finally **A** outputs a bit b.

Fig. 1. Real Game: $\mathtt{Game}_{real}$

3. The client verifies the validity of $(\mathtt{C}(w), Tag)$. If he accepts, then he decrypts each $C_i \in \mathtt{C}(w)$, and outputs $\mathtt{D}(w) = \{D_i \mid D_i \text{ contains } w\}$. Otherwise he outputs `reject`.

The definition of usual searchable symmetric encryption (SSE) schemes [10, 11] is obtained by deleting Tag from the verifiable SSE schemes.

2.2 Privacy

Suppose that the server (who is an adversary **A**) is honest but curious. In any SSE scheme, the server learns $|D_1|, \cdots, |D_n|$ and $|\mathcal{W}|$ in the store phase. Also in the search phase, she learns $\mathtt{List}(w) = \{i \mid D_i \text{ contains } w\}$ for the search keyword w because she must be able to return $\mathtt{C}(w)$. Now the server should not be able to learn any more information. Curtmola, Garay, Kamara and Ostrovsky [10, 11] formulated this security notion as follows.

We consider a real game $\mathtt{Game}_{real}$ and a simulation game $\mathtt{Game}_{sim}$. $\mathtt{Game}_{real}$ is played by a challenger and an adversary **A** as shown in Fig.1. $\mathtt{Game}_{sim}$ is played by a challenger, an adversary **A** and a simulator **Sim** as shown in Fig.2.

Let

$$p_0 = \Pr(\mathbf{A} \text{ outputs } b = 1 \text{ in } \mathtt{Game}_{real}),$$
$$p_1 = \Pr(\mathbf{A} \text{ outputs } b = 1 \text{ in } \mathtt{Game}_{sim}).$$

Definition 1. *We say that a (verifiable) SSE scheme satisfies privacy if there exists a PPT simulator* **Sim** *such that $|p_0 - p_1|$ is negligible for any PPT adversary* **A**.

2.3 Reliability (Verifiability)

Suppose that the server (who is an adversary **A**) is malicious. In verifiable SSE schemes, the server should not be able to forge a search result $(\mathtt{C}(w), Tag)$ in the search phase. This security notion is formulated as follows [21].

Fix $(\mathcal{D}, \mathcal{W}, \mathtt{Index})$ and search queries $w_1, \cdots, w_q \in \mathcal{W}$ arbitrarily. We say that **A** wins if she can return $(\mathtt{C}(w_i)^*, Tag^*)$ for some query $t(w_i)$ such that $\mathtt{C}(w_i)^* \neq \mathtt{C}(w_i)$ and the client accepts $(\mathtt{C}(w_i)^*, Tag^*)$.

Simulation Game ($\mathtt{Game}_{sim}$)

In the store phase,

- **A** chooses $(\mathcal{D}, \mathcal{W}, \mathtt{Index})$ and sends them to the challenger.
- The challenger sends $|D_1|, \cdots, |D_n|$ and $|\mathcal{W}|$ to simulator **Sim**, where $\mathtt{D} = \{D_1, \cdots, D_n\}$.
- **Sim** returns $(\mathcal{I}', \mathtt{C}')$ to the challenger, and he replays them to **A**.

In the search phase, for $i = 1, \cdots, q$,

1. **A** chooses a keyword $w_{a_i} \in \mathcal{W}$ and sends it to the challenger.
2. The challenger sends $\mathtt{List}(w_{a_i}) = \{j \mid D_j \text{ contains } w_{a_i}\}$ to **Sim**.
3. **Sim** returns t' to the challenger, and he relays it to **A**.

Finally **A** outputs a bit b.

Fig. 2. Simulation Game: $\mathtt{Game}_{sim}$

Definition 2. *We say that a verifiable SSE satisfies reliability if for any PPT adversary* **A***,* $\Pr(\mathbf{A}$ *wins*$)$ *is negligible for any* $(\mathcal{D}, \mathcal{W}, \mathtt{Index})$ *and any search queries* $w_1, \cdots, w_q$*.*

Kurosawa et al. [21] proved the following proposition.

Proposition 1. *A verifiable SSE scheme satisfies privacy and reliability if and only if the corresponding protocol is UC-secure against non-adaptive adversaries.*

3 Our Efficient Verifiable SSE Scheme

In this section, we show a more efficient verifiable SSE scheme than the previous one [21]. In this scheme, the client sends only $n + 128$ bits in the search phase while $(\log n + \ell + 1) \times n$ bits must be sent in [21], where n is the number of documents and ℓ is the bit length of each keyword.

Remember that $\mathcal{D} = \{D_1, \cdots, D_n\}$ is a set of documents, $\mathcal{W} = \{w_1, \cdots, w_m\}$ is a set of keywords and $\mathtt{Index} = \{e_{i,j}\}$ is an $m \times n$ binary matrix such that

$$e_{i,j} = \begin{cases} 1 \text{ if } w_i \text{ is contained in } D_j \\ 0 \; otherwise \end{cases}.$$

Let $\mathtt{index}_i$ denote the ith row of $\mathtt{Index}$.

3.1 Our Efficient SSE Scheme

In this subsection, we assume that the server is honest but curious. Let $\mathtt{PRF}_k : \{0,1\}^\ell \times \{0,1\}^* $ be a pseudorandom function, where k is a key. Let $\mathtt{SKE} = (G, E, E^{-1})$ be a symmetric-key encryption scheme, where G is a key generation algorithm, E is an encryption algorithm and E^{-1} is a decryption algorithm. We assume that $\mathtt{SKE}$ is CPA-secure in the left-or right sense [2].

Now our SSE scheme is as follows.

(Store phase)

1. The client generates (k_e, k_0, k_1) randomly, where k_e is a key of $\mathtt{SKE}$, and k_0, k_1 are keys of $\mathtt{PRF}$. He then keeps (k_e, k_0, k_1) secret.
2. The client computes $C_i = E_{k_e}(D_i)$ for each document $D_i \in \mathcal{D}$. He also computes

$$\mathtt{label}_i = [\mathtt{PRF}_{k_0}(w_i)]_{1..128}$$
$$\overline{\mathtt{index}}_i = \mathtt{index}_i \oplus [\mathtt{PRF}_{k_1}(w_i)]_{1..n}$$

for each keyword $w_i \in \mathcal{W}$. He also chooses a random permutation σ on $\{1, \cdots, m\}$. He then stores

$$\mathcal{C} = (C_1, \cdots, C_n) \text{ and } \mathcal{I} = \{(\mathtt{label}_{\sigma(i)}, \overline{\mathtt{index}}_{\sigma(i)}) \mid i = 1, \cdots, m\}$$

to the server.

(Search phase) Suppose that the client wants to search on a keyword w_a.

1. The client computes $\mathtt{label}_a$ and $\mathtt{pad}_a = [\mathtt{PRF}_{k_1}(w_a)]_{1..n}$. He then sends $t(w_a) = (\mathtt{label}_a, \mathtt{pad}_a)$ to the server.
2. The server finds $(\mathtt{label}_a, \overline{\mathtt{index}}_a) \in \mathcal{I}$ by using $\mathtt{label}_a$. She then computes

$$\mathtt{index}_a = \overline{\mathtt{index}}_a \oplus \mathtt{pad}_a$$

Let $\mathtt{index}_a = (e_1, \cdots, e_n)$. She returns $\mathtt{C}(w) = \{C_i \mid e_i = 1\}$ to the client.
3. The client decrypts all C_i such that $C_i \in \mathtt{C}(w)$, and outputs $\{D_i \mid C_i \in \mathtt{C}(w)\}$.

Suppose that there are 5 documents $\mathcal{D} = \{D_1, \cdots, D_5\}$ and 2 keywords $\mathcal{W} = \{w_1, w_2\}$ such that $D(w_1) = \{D_1, D_3, D_5\}$ and $D(w_2) = \{D_2, D_4\}$. Then

$$\overline{\mathtt{index}}_1 = (1, 0, 1, 0, 1) \oplus [\mathtt{PRF}_{k_1}(w_1)]_{1..5}$$
$$\overline{\mathtt{index}}_2 = (0, 1, 0, 1, 0) \oplus [\mathtt{PRF}_{k_1}(w_2)]_{1..5}$$

Theorem 1. *The above scheme satisfies privacy if* $\mathtt{SKE}$ *is CPA-secure and* $\mathtt{PRF}$ *is a pseudorandom function.*

Proof. (Sketch) In $\mathtt{Game}_{sim}$, our simulator **Sim** behaves as follows.

(Store phase) **Sim** receives $|D_1|, \cdots, |D_n|$ and $m = |\mathcal{W}|$ from the challenger.

1. **Sim** generates a key k_e of $\mathtt{SKE}$ randomly. It also chooses a random permutation σ on $\{1, \cdots, m\}$.
2. **Sim** computes $C_i = E_{k_e}(0^{|D_i|})$ for $i = 1, \cdots, n$. **Sim** also chooses $\mathtt{label}_i \in \{0, 1\}^{128}$ and $\overline{\mathtt{index}}_i \in \{0, 1\}^n$ randomly for $i = 1, \cdots, m$.

3. Finally **Sim** returns $\mathcal{C}' = (C_1, \cdots, C_n)$ and $\mathcal{I}' = \{(\mathtt{label}_{\sigma(i)}, \overline{\mathtt{index}}_{\sigma(i)}) \mid i = 1, \cdots, m\}$ to the challenger.

(Search phase) **Sim** receives $\mathtt{List}(w_{a_i}) = \{j \mid D_j \text{ contains } w_{a_i}\}$ from the challenger for $i = 1, \cdots, q$. For each i, let

$$e_j = \begin{cases} 1 \text{ if } j \in \mathtt{List}(w_{a_i}) \\ 0 \ otherwise \end{cases} .$$

Sim then computes $\mathtt{pad}^* = \overline{\mathtt{index}}_{\sigma(i)} \oplus (e_1, \cdots, e_n)$ and returns $t' = (\mathtt{label}_{\sigma(i)}, \mathtt{pad}^*)$ to the challenger.

Now the adversary **A** has $(\mathcal{D}, \mathcal{W}, \mathtt{Index})$. Still in the store phase, **A** cannot distinguish $\mathcal{C}'$ from $\mathcal{C}$ because SKE is CPA-secure. Also **A** cannot distinguish $\mathcal{I}'$ from $\mathcal{I}$ because PRF (which is used in $\mathtt{Game}_{real}$) is a pseudorandom function.

In the search phase, **A** cannot distinguish $t' = (\mathtt{label}_{\sigma(i)}, \mathtt{pad}^*)$ from $t(w_a) = (\mathtt{label}_a, \mathtt{pad}_a)$ because PRF is a pseudorandom function and σ is a random permutation. Therefore **A** cannot distinguish $\mathtt{Game}_{sim}$ from $\mathtt{Game}_{real}$. □

3.2 Our Efficient Verifiable SSE Scheme

In this subsection, we assume that the server is malicious, and extend the above SSE scheme to a verifiable SSE scheme. (It is more efficient than the previous verifiable SSE scheme [21].) Let $\mathtt{MAC}_{k_m}$ be a tag generation algorithm of MAC, where k_m is a key. We assume that MAC is a pseudorandom function. (This means that it is unforgeable against chosen message attack.)

For keyword w_1, a malicious server may return $(\underline{C_2}, C_3, C_5)$ instead of (C_1, C_3, C_5). A naive approach to prevent such active attacks would be to replace each C_i with $(C_i, \mathtt{MAC}_{k_m}(C_i))$. However, this method does not work because $(\underline{C_2}, \mathtt{MAC}_{k_m}(\underline{C_2}))$ is a valid pair. In our verifiable SSE scheme, the server returns $\mathtt{MAC}_{k_m}(\mathtt{label}_1, (C_1, C_3, C_5))$. This method can prevent the above attack because the server must forge
$\mathtt{MAC}_{k_m}(\mathtt{label}_1, (\underline{C_2}, C_3, C_5))$.

Now our verifiable SSE scheme is obtained by modifying the SSE scheme of Sec.3.1 as follows.

(Store phase)

1' The client generates a MAC key k_m randomly, and keeps it secret together with (k_e, k_0, k_1).
2' The client computes $tag_i = \mathtt{MAC}_{k_m}(\mathtt{label}_i, C(w_i))$ for each keyword $w_i \in \mathcal{W}$, and stores

$$\mathcal{I} = \{(\mathtt{label}_{\sigma(i)}, \overline{\mathtt{index}}_{\sigma(i)}, tag_{\sigma(i)}) \mid i = 1, \cdots, m\} \tag{2}$$

to the server, where $\mathtt{label}_i$ and $\overline{\mathtt{index}}_i$ are computed in the same way as in Sec.3.1, and σ is a random permutation on $\{1, \cdots, m\}$.

(Search phase) Suppose that the client wants to search on a keyword w_a.

1’ The client sends $(\mathtt{label}_a, \mathtt{pad}_a)$ to the server in the same way as in Sec.3.1.
2’ The server finds $(\mathtt{label}_a, \overline{\mathtt{index}}_a, tag_a) \in \mathcal{I}$ by using $\mathtt{label}_a$. She then returns tag_a and $\mathtt{C}(w)$ to the client.
3’ If $tag_a = \mathtt{MAC}_{k_m}(\mathtt{label}_a, \mathtt{C}(w))$, then the client decrypts all C_i such that $C_i \in \mathtt{C}(w)$, and outputs them. Otherwise he outputs $\mathtt{reject}$.

In the example of Sec.3.1,

$$tag_1 = \mathtt{MAC}_{k_m}(\mathtt{label}_1, (C_1, C_3, C_5)),\ tag_2 = \mathtt{MAC}_{k_m}(\mathtt{label}_2, (C_2, C_4)),$$

Theorem 2. *The above scheme satisfies privacy and reliability if* SKE *is CPA-secure, and* PRF *and* MAC *are pseudorandom functions.*

Proof. (Sketch) We can prove the privacy similarly to the proof of Theorem 1. Hence will will prove the reliability.

Suppose that there exists an adversary **A** who breaks the reliability for some $(\mathcal{D}, \mathcal{W}, \mathtt{Index})$ and some search queries $w_1, \cdots, w_q$. We will show a forger **B** for the underlying MAC. **B** runs **A** by playing the role of a client with $(\mathcal{D}, \mathcal{W}, \mathtt{Index})$ and $w_1, \cdots, w_q$ as an input.

In the store phase, to compute $\mathcal{I}$, **B** obtains each $tag_i = \mathtt{MAC}_{k_m}(\mathtt{label}_i, C(w_i))$ from his MAC oracle, where k_m is randomly chosen by the MAC oracle. That is, for $i = 1, \cdots, q$, **B** queries $(\mathtt{label}_i, C(w_i))$ to the MAC oracle, and receives tag_i.

In the search phase, if **A** returns $(\mathtt{C}(w_i)^*, tag_i^*)$ such that $\mathtt{C}(w_i)^* \neq \mathtt{C}(w_i)$ for some $(\mathtt{label}_i, \mathtt{pad}_i)$, then **B** outputs $(\mathtt{label}_i, C(w_i)^*)$ and tag_i^* as a forgery.

From our assumption, **A** returns such $(\mathtt{C}(w_i)^*, tag_i^*)$ with non-negligible probability. It also holds that

$$tag_i^* = \mathtt{MAC}_{k_m}(\mathtt{label}_i, C(w_i)^*)$$

with non-negligible probability from our assumption. Finally note that **B** never queried $(\mathtt{label}_i, C(w_i)^*) \neq (\mathtt{label}_i, C(w_i))$ to the MAC oracle.

Therefore **B** succeeds in forgery with non-negligible probability. This is against our assumption on MAC. Hence our scheme satisfies reliability. □

4 How to Update Documents

4.1 Our Idea

In the scheme of Sec.3.2, the client stores $tag_1 = \mathtt{MAC}_{k_m}(\mathtt{label}_1, (C_1, C_3, C_5))$ for a keyword w_1. In this scheme, however, the client cannot modify each C_i efficiently. For example, suppose that C_1 includes two keywords, w_1 and w_2. To modify C_1 to C_1', the client must store two updated authenticators, $\mathtt{MAC}(\mathtt{label}_1, (C_1', C_3, C_5))$ and $\mathtt{MAC}(\mathtt{label}_2, (C_1', \cdots))$, to the server in the update phase. If C_1 includes more keywords, then the client must updates more authenticators.

Now our idea is that the client authenticates only $(\mathtt{label}_1, 1, 3, 5)$. He separately authenticates each (i, C_i) also. Then to update C_1 to C_1', the client stores just an authenticator on $(1, C_1')$. The update cost is only this no matter how many keywords are included in C_1. Thus the client can update each C_i efficiently.

To delete a document C_1, the client updates it to a special symbol $C_1' = \mathit{delete}$ similarly. To add a new document D_6 which includes w_1, the client updates the authenticator on $(\texttt{label}_1, 1, 3, 5)$ to that on $(\texttt{label}_1, 1, 3, 5, 6)$.

4.2 How to Time Stamp

The last problem is how to times tamp on the current (i, C_i), and how to time stamp on the current/updated $(\texttt{label}_1, 1, 3, 5, 6)$.

We can solve this problem by using an authentication scheme which posses the timestamp functionality such as Merkle hash tree [24], or authenticated skiplist [18] or the RSA accumulator [5, 14]. Such a scheme allows one to hash a set of inputs into one short accumulation value, such that there is a witness that a given input was incorporated into the accumulator, and at the same time, it is infeasible to find a witness for a value that was not accumulated.

The size of witness is $O(\log n)$ in the Merkle hash tree and the authenticated skiplist, where n is the number of documents. It is $O(\lambda)$ in the RSA accumulator, where λ is the security parameter. We can use any one of them. In what follows, we present our scheme based on the RSA accumulator.

4.3 RSA Accumulator

Let $p = 2p' + 1$ and $q = 2q' + 1$ be two large primes such that p' and q' are also primes and $|pq| > 3\lambda$. Let $N = pq$ and let

$$QR_N = \{a \mid a = x^2 \bmod N \text{ for some } x \in Z_N^*\}.$$

Then QR_N is a cyclic group of size $(p-1)(q-1)/4$. Let g be a generator of QR_N. We say that a family of functions $F = \{f : A \to B\}$ is two-universal if $Pr[f(x_1) = f(x_2)] = 1/|B|$ for all $x_1 \neq x_2$ and for a randomly chosen function $f \in F$.

Proposition 2. *[16] For any $y \in \{0,1\}^\lambda$, we can compute a prime $x \in \{0,1\}^{3\lambda}$ such that $f(x) = y$ by sampling $O(\lambda^2)$ times with overwhelming probability from the set of inverses $f^{-1}(y)$, where the probability is taken over $f \in F$.*

Let $F = \{f_a : \{0,1\}^{3\lambda} \to \{0,1\}^\lambda\}$ be a two-universal family of functions and choose $f \in F$ randomly. (Such functions can be built easily. For instance, view a and x as members of $GF(2^{3\lambda})$, and let $f_a(x)$ be the λ least significant bits of $a \times x$.)

For a set $E = \{y_1, \cdots, y_n\}$ with $y_i \in \{0,1\}^\lambda$, the RSA accumulator works as follows.

1. For each y_i, Alice chooses a prime x_i such that $f(x_i) = y_i$ randomly. Let $prime(y_i)$ denote such a prime x_i. She then computes the accumulated value of $E = \{y_1, \cdots, y_n\}$ as
$$\texttt{Acc}(E) = g^{\prod_{i=1}^n prime(y_i)} \bmod N$$
and sends $\texttt{Acc}(E)$ to Bob.

2. Later Alice proves that $y_j \in E$ to Bob as follows. She computes

$$\pi_j = g^{\prod_{i \neq j} prime(y_i)} \bmod N$$

and sends π_j and $prime(y_j)$ to Bob.
3. Bob verifies that

$$\texttt{Acc}(E) = (\pi_j)^{prime(y_j)} \bmod N.$$

Definition 3. *[6] (Strong RSA assumption) Given $N = pq$ and a random element $y \in Z_N$, it is hard to find x and $e > 1$ such that $y = x^e \bmod N$.*

Proposition 3. *Given N, g, f and $E = \{y_1, \cdots, y_n\}$, it is hard to find $y \notin E$ and π such that*

$$\pi^{prime(y)} = \texttt{Acc}(E) \bmod N \tag{3}$$

under the strong RSA assumption.

If we want to apply the above protocol to a set $A = \{a_1, \cdots, a_n\}$ with $a_i \notin \{0,1\}^\lambda$ for some i, then we define the accumulated value of A as

$$\texttt{Acc}(A) = g^{\prod_{i=1}^{n} prime(H(a_i))} \bmod N,$$

where $H : \{0,1\}^* \to \{0,1\}^\lambda$ is a collision resistant hash function. Namely we apply the above protocol to the set $\{H(a_1), \cdots, H(a_n)\}$.

Note that $prime(H(a_i))$ is a prime $x_i \in \{0,1\}^{3\lambda}$ such that $f(x_i) = H(a_i)$, where $f : \{0,1\}^{3\lambda} \to \{0,1\}^\lambda$ is a two-universal hash function. We can compute such a prime x_i efficiently for any $H(a_i) \in \{0,1\}^\lambda$ from Proposition 2.

5 Proposed Verifiable Dynamic SSE Scheme

In this section, we show the details of our idea, i.e., how to *modify*, *delete* and *add* documents efficiently in a verifiable SSE scheme, where the server is a malicious adversary. We call such a scheme a verifiable dynamic SSE scheme.

5.1 Scheme

In the proposed scheme,

– The client applies the RSA accumulator to the sets

$$E_C = \{(i, C_i) \mid i = 1, \cdots, n\},$$
$$E_I = \{(\texttt{label}_i, j, [\overline{\texttt{index}}_i]_j) \mid i = 1, \cdots, m, j = 1, \cdots, n\},$$

and compute their accumulated values $\texttt{Acc}(E_C)$ and $\texttt{Acc}(E_I)$.
– He updates $\texttt{Acc}(E_C)$ each time when he modifies or deletes a document, and updates $\texttt{Acc}(E_I)$ each time when he adds a document.
– In the search phase, the client checks if a server returned the valid (updated) ciphertexts based on $\texttt{Acc}(E_C)$ and $\texttt{Acc}(E_I)$.

A subtle problem is how the client and the server compute the same $prime(y)$ locally, where $y = (i, C_i)$ or $(\mathtt{label}_i, j, [\overline{\mathtt{index}}_i]_j)$. Remember that $prime(y)$ is a prime x such that $f(x) = y$. and such x is chosen *randomly*. In the proposed scheme, the client chooses k_a randomly, and sends it to the server at the beginning of the protocol. Then they use $\mathtt{PRF}_{k_a}(y)$ as the randomness when computing $prime(y)$. Thus they can compute the same $prime(y)$ locally.

Let $F = \{f : \{0,1\}^{3\lambda} \to \{0,1\}^{\lambda}\}$ be a two-universal family of functions, and $H : \{0,1\}^* \to \{0,1\}^{\lambda}$ be a collision-resistant hash function. Let $[\overline{\mathtt{index}}_i]_j$ denote the jth bit of $\overline{\mathtt{index}}_i$.

(Store phase)

1. The client generates $(N(= pq), g)$ as shown in Sec. 4.3 and chooses $f \in F$ randomly. He also generates (k_e, k_0, k_1, k_a) randomly, where k_e is a key of SKE, and k_0, k_1, k_a are keys of PRF. He further chooses a random permutation σ on $\{1, \cdots, m\}$. He then sends (N, g, f, k_a) to the server and keeps $(p, q, k_e, k_0, k_1, \sigma)$ secret.
2. The client computes $C_i = E_{k_e}(D_i)$ for each document $D_i \in \mathcal{D}$. He also computes

$$\mathtt{label}_i = [\mathtt{PRF}_{k_0}(w_i)]_{1..128},\ \mathtt{pad}_i = [\mathtt{PRF}_{k_1}(w_i)]_{1..n},\ \overline{\mathtt{index}}_i = \mathtt{pad}_i \oplus (e_{i,1}, \cdots, e_{i,n})$$

for each keyword $w_i \in \mathcal{W}$. He then stores $\mathcal{C} = (C_1, \cdots, C_n)$ and

$$\mathcal{I} = \{(\mathtt{label}_{\sigma(i)}, \overline{\mathtt{index}}_{\sigma(i)}) \mid i = 1, \cdots, m\} \tag{4}$$

to the server.
3. He also computes

$$A_C = g^{\prod_{i=1}^{n} prime(H(i,H(C_i)))} \bmod N,$$
$$A_I = g^{\prod_{i=1}^{m} \prod_{j=1}^{n} prime(H(\mathtt{label}_i, j, [\overline{\mathtt{index}}_i]_j))} \bmod N.$$

He then keeps n, A_C and A_I.

(Search phase) Suppose that the client wants to search on a keyword w_a.

1. The client computes $(\mathtt{label}_a, \mathtt{pad}_a)$ and sends them to the server.
2. The server finds $(\mathtt{label}_a, \overline{\mathtt{index}}_a) \in \mathcal{I}$ by using $\mathtt{label}_a$. She computes

$$(e_1, \cdots, e_n) = \mathtt{pad}_a \oplus \overline{\mathtt{index}}_a$$

and sets $\mathtt{C}'(w) = \{(i, C_i) \mid e_i = 1\}$. She next computes

$$\pi_C = g^{\prod_{e_i=0} prime(H(i,H(C_i)))} \bmod N,$$
$$\pi_I = g^{\prod_{i \neq a} \{\prod_{j=1}^{n} prime(H(\mathtt{label}_i, j, [\overline{\mathtt{index}}_i]_j))\}} \bmod N.$$

Finally she returns $(\mathtt{C}'(w), \pi_C, \pi_I)$ to the client.

3. The client first computes $x_i = prime(H(i, H(C_i)))$ for each $(i, C_i) \in \mathtt{C}'(w)$, and checks if

$$A_C = (\pi_C)^{\prod_{e_i=1} x_i} \bmod N \tag{5}$$

The client next reconstructs $(e_1, \cdots, e_n)$ from $\mathtt{C}'(w)$ and computes $\overline{\mathtt{index}}_a = \mathtt{pad}_a \oplus (e_1, \cdots, e_n)$. He then computes $z_j = prime(H(\mathtt{label}_a, j, [\overline{\mathtt{index}}_a]_j))$ for $j = 1, \cdots, n$, and checks if

$$A_I = (\pi_I)^{\prod_{j=1}^{n} z_j} \bmod N \tag{6}$$

If all the checks succeed, then the client decrypts all C_i such that $e_i = 1$ and outputs the documents $\{D_i \mid e_i = 1\}$. Otherwise he outputs `reject`.

(Remark.)

- Eq.(5) verifies the correctness of $\mathtt{C}'(w_a) = \{(i, C_i) \mid D_i \text{ contains } w_a\}$. Eq.(6) verifies the correctness of $\overline{\mathtt{index}}_a$. Hence it verifies the correctness of $(e_1, \cdots, e_n)$.
- For example, if both $(e_1, \cdots, e_5) = (1, 0, 1, 0, 1)$ and $(1, C_1), (3, C_3), (5, C_5)$ are valid, then it is clear that (C_1, C_3, C_5) are the correct ciphertexts.

(Modify) Suppose that the client wants to modify C_i to C_i'.

1. The client send (i, C_i') to the server.
2. The server computes

$$\pi_i = g^{\prod_{j \neq i} prime(H(j, H(C_j)))} \bmod N$$

and returns $(H(C_i), \pi_i)$ to the client.

3. The client computes $x_i = prime(H(i, H(C_i)))$ and checks if

$$A_C = (\pi_i)^{x_i} \bmod N. \tag{7}$$

If the check fails, then he outputs `reject`. Otherwise he computes

$$\begin{aligned} x_i' &= prime(H(i, H(C_i'))), \\ d &= x_i'/x_i \bmod (p-1)(q-1), \\ A_C' &= (A_C)^d = g^{x_1 \cdots x_i' \cdots x_n} \bmod N. \end{aligned}$$

He finally updates A_C to A_C'.

(Delete) Suppose that the client wants to delete C_i. He frist sends $(i, delete)$ to the server. Then apply (Modify) to $C_i' = delete$.

(Add) Suppose that the client wants to add a document D_{n+1}. Let

$$e_{i,n+1} = \begin{cases} 1 \text{ if } w_i \text{ is contained in } D_{n+1} \\ 0 \; otherwise \end{cases} . \tag{8}$$

1. The client computes $C_{n+1} = E_{k_e}(D_{n+1})$, and sends C_{n+1} to the server. He also updates A_C to
$$A'_C = (A_C)^{prime(H(n+1,H(C_{n+1})))} \bmod N.$$
2. The client also computes $a_i = [\mathtt{PRF}_{k_1}(w_i)]_{n+1} \oplus e_{i,n+1}$ for $i = 1, \cdots, m$, where $[\mathtt{PRF}_{k_1}(w_i)]_{n+1}$ denotes the $(n+1)$th bit of $\mathtt{PRF}_{k_1}(w_i)$.
He then sends $(a_{\sigma(1)}, \cdots, a_{\sigma(m)})$ to the server.
3. The server updates $\overline{\mathtt{index}}_{\sigma(i)}$ to $\overline{\mathtt{index}}'_{\sigma(i)} = \overline{\mathtt{index}}_{\sigma(i)} || a_{\sigma(i)}$ for $i = 1, \cdots, m$, where $||$ denotes concatenation.
4. The client computes $z_i = prime(H(\mathtt{label}_i, n+1, a_i))$ for $i = 1, \cdots, m$, and updates A_I to
$$A'_I = (A_I)^{z_1 \cdots z_m} \bmod N.$$
Finally he updates n to $n+1$.

5.2 Example

Consider the example shown in Sec.3.1. In the store phase, the client computes

$$A_C = g^{\prod_{i=1}^{5} prime(H(i,H(C_i)))} \bmod N,$$
$$A_I = g^{\prod_{i=1}^{2} \prod_{j=1}^{5} prime(H(\mathtt{label}_i, j, [\overline{\mathtt{index}}_i]_j))} \bmod N$$

and keeps $n = 5, A_C$ and A_I.

(Search phase) Suppose that the client wants to search on w_1. He then sends $(\mathtt{label}_1, \mathtt{pad}_1)$ to the server.

1. The server finds $\overline{\mathtt{index}}_1$ from $\mathcal{I}$, and computes $\mathtt{pad}_1 \oplus \overline{\mathtt{index}}_1 = (1, 0, 1, 0, 1)$. From this $(1, 0, 1, 0, 1)$, she sets $\mathtt{C}'(w_1) = \{(1, C_1), (3, C_3), (5, C_5)\}$. She then computes
$$\pi_C = g^{\prod_{i=2,4} prime(H(i,H(C_i)))} \bmod N,$$
$$\pi_I = g^{\prod_{j=1}^{5} prime(H(\mathtt{label}_2, j, [\overline{\mathtt{index}}_2]_j))} \bmod N.$$
Finally she returns $(\mathtt{C}'(w_1), \pi_C, \pi_I)$ to the client.
2. The client computes $x_i = prime(H(i, H(C_i)))$ for $i = 1, 3, 5$, and checks if
$$A_C = (\pi_C)^{\prod_{i=1,3,5} x_i} \bmod N. \tag{9}$$
Also he reconstructs $\overline{\mathtt{index}}_1 = \mathtt{pad}_1 \oplus (1, 0, 1, 0, 1)$ from $\mathtt{C}'(w_1)$. He then computes $z_j = prime(H(\mathtt{label}_1, j, [\overline{\mathtt{index}}_1]_j))$ for $j = 1, \cdots, 5$, and checks if
$$A_I = (\pi_I)^{\prod_{j=1}^{5} z_j} \bmod N. \tag{10}$$
If all the checks succeed, then the client decrypts (C_1, C_3, C_5), and outputs the documents (D_1, D_3, D_5). Otherwise he outputs `reject`.

(Modify) Suppose that the client wants to modify C_1 to C_1'.

1. The client sends $(1, C_1')$ to the server.
2. The server computes
$$\pi_1 = g^{\prod_{j=2}^{5} prime(H(j,H(C_j)))} \bmod N$$
and returns $(H(C_1), \pi_1)$ to the client.
3. The client computes $x_1 = prime(H(1, H(C_1)))$ and checks if
$$A_C = (\pi_1)^{x_1} \bmod N.$$
If the check fails, then he outputs `reject`. Otherwise he computes
$$x_1' = prime(H(1, H(C_1'))),$$
$$d = x_1'/x_1 \bmod (p-1)(q-1),$$
$$A_C' = (A_C)^d = g^{x_1' x_2 \cdots x_5} \bmod N.$$
He finally updates A_C to A_C'.

(Delete) Suppose that the client wants to delete C_2. He first sends $(2, delete)$ to the server. Then apply (Modify) to $C_2' = delete$.

(Add) Suppose that the client wants to add a document D_6 which contains w_1 as a keyword.

1. The client computes $C_6 = E_{k_e}(D_6)$, and sends C_6 to the server. He also updates A_C to $A_C' = (A_C)^{prime(H(6,H(C_6)))} \bmod N$.
2. The client also computes $a_1 = [\mathtt{PRF}_{k_1}(w_1)]_6 \oplus 1$ and $a_2 = [\mathtt{PRF}_{k_1}(w_2)]_6 \oplus 0$. He then sends $(a_{\sigma(1)}, a_{\sigma(2)})$ to the server.
3. The server updates $\overline{\mathtt{index}}_{\sigma(i)}$ to $\overline{\mathtt{index}}'_{\sigma(i)} = \overline{\mathtt{index}}_{\sigma(i)} || a_{\sigma(i)}$ for $i = 1, 2$.
4. The client computes $z_i = prime(H(\mathtt{label}_i, 6, a_i))$ for $i = 1, 2$, and updates A_I to $A_I' = (A_I)^{z_1 \cdot z_2} \bmod N$. Finally he updates $n = 5$ to $n = 6$.

6 Security

In this section, we prove that the proposed verifiable dynamic SSE scheme is UC-secure. If a protocol Σ is secure in the universally composable (UC) security framework, its security is maintained under a general protocol composition [7–9].

In the UC framework, there exists an environment $\mathcal{Z}$ which generates the input to all parties, reads all outputs, and in addition interacts with an adversary $\mathbf{A}$ in an arbitrary way throughout the computation.

A protocol Σ is said to securely realize a given functionality $\mathcal{F}$ if for any adversary $\mathbf{A}$, there exists an ideal world adversary $\mathbf{S}$ such that no environment $\mathcal{Z}$ can tell whether it is interacting with $\mathbf{A}$ and parties running the protocol, or with $\mathbf{S}$ and parties that interact with $\mathcal{F}$ in the ideal world.

6.1 Ideal Functionality

We describe the ideal functionality $\mathcal{F}$ of verifiable dynamic SSE schemes in Fig.3. In the ideal world, $\mathcal{Z}$ interacts with the dummy client and the dummy server, where the dummy players communicate with $\mathcal{F}$.

Our $\mathcal{F}$ provides an ideal world because the ideal world adversary $\mathbf{S}$ (i.e., a malicious server) learns only $|D_1|, \cdots, |D_n|$ and $|\mathcal{W}|$ for the store command of $\mathcal{Z}$, only $\mathtt{List}(w)$ for a search command on keyword w, only $(i, |D_i'|)$ for a modify command on (i, D_i'), only i for a delete command on i, and only $|D|$ for an add command on D. (See the beginning of Sec.2.2.)

We say that a protocol (client, server) is UC-secure if it securely realizes the ideal functionality $\mathcal{F}$.

Ideal Functionality $\mathcal{F}$

Running with the dummy client P_1, the dummy server P_2 and an adversary $\mathbf{S}$.

- Upon receiving $(\mathbf{store}, sid, \mathcal{D}, \mathcal{W}, \mathtt{Index})$ from P_1, verify that this is the first input from P_1 with $(\mathbf{store}, sid)$. If so, store $(n, \mathcal{D}, \mathcal{W}, \mathtt{Index})$, and send $|D_1|, \cdots, |D_n|$ and $|\mathcal{W}|$ to $\mathbf{S}$. Otherwise ignore this input.
- Upon receiving $(\mathbf{search}, sid, w_a)$ from P_1, send $\mathtt{List}(w_a)$ to $\mathbf{S}$, where $w_a \in \mathcal{W}$.
  1. If $\mathbf{S}$ returns OK, then send $\mathtt{D}(w_a)$ to P_1.
  2. If $\mathbf{S}$ returns $\mathtt{reject}$, then send $\mathtt{reject}$ to P_1.
- Upon receiving $(\mathbf{modify}, sid, i, D_i')$ from P_1, send $(i, |D_i'|)$ to $\mathbf{S}$.
  1. If $\mathbf{S}$ returns OK, then replace D_i with D_i'.
  2. If $\mathbf{S}$ returns $\mathtt{reject}$, then send $\mathtt{reject}$ to P_1.
- Upon receiving $(\mathbf{delete}, sid, i)$ from P_1, send i to $\mathbf{S}$.
  1. If $\mathbf{S}$ returns OK, then let $D_i := \mathit{delete}$.
  2. If $\mathbf{S}$ returns $\mathtt{reject}$, then send $\mathtt{reject}$ to P_1.
- Upon receiving $(\mathbf{add}, sid, D)$ from P_1, add D to $\mathcal{D}$, and send $|D|$ to $\mathbf{S}$.

Fig. 3. Ideal Functionality of Dynamic SSE

6.2 UC-Security of Our Scheme

Theorem 3. *The proposed scheme is UC-secure against non-adaptive adversaries under the strong RSA assumption if* SKE *is CPA-secure,* PRF *is a pseudo-random function and H is a collision-resistant hash function.*

A proof is given in Appendix A.

7 Efficiency

7.1 Efficiency of the Proposed Verifiable Dynamic SSE Scheme

Table 2 shows the communication overheads and the computation costs of the proposed verifiable dynamic SSE scheme. For example, in the search phase, to search on a keyword w_a, the client sends $(\texttt{label}_a, \texttt{pad}_a)$ to the server, and the server returns $(\texttt{C}'(w), \pi_C, \pi_I)$, where $\texttt{C}'(w) = \{(i, C_i) \mid D_i \text{ contains } w\}$. Therefore the total communication cost is

$$T_s = |\texttt{label}_a| + |\texttt{pad}_a| + |\texttt{C}'(w)| + |\pi_C| + |\pi_I|.$$

Hence the communication overhead is

$$T_s - |\texttt{C}'(w)| = |\texttt{label}_a| + |\texttt{pad}_a| + |\pi_C| + |\pi_I| = n + O(\lambda),$$

where λ is the security parameter of the RSA accumulator.

Table 2. Efficiency of the Proposed Verifiable Dynamic SSE Scheme

	search	modify	delete	add
communication overhead	$n + O(\lambda)$	$O(\lambda)$	$O(\lambda)$	m
computation cost of the server	$O(nm)$	$O(n)$	$O(n)$	$O(m)$
computation cost of the client	$O(n)$	$O(1)$	$O(1)$	$O(m)$

The storage overhead is $n(m + 128)$.

7.2 More Efficient Variant with No *Add*

Suppose that the client does not add new documents. Then we can consider a more efficient variant of the proposed scheme such that the RSA accumulator is not used to authenticate $\texttt{Index}$.

Instead, the client computes $tag_i = \texttt{MAC}_{k_m}(\texttt{label}_i, \texttt{List}(w_i))$ for each keyword $w_i \in \mathcal{W}$, and stores

$$\mathcal{I} = \{(\texttt{label}_{\sigma(i)}, \overline{\texttt{index}}_{\sigma(i)}, tag_{\sigma(i)}) \mid i = 1, \cdots, m\} \tag{11}$$

to the server in the store phase.

In the search phase, the server returns tag_a to the client for a search keyword w_a instead of π_I. Then the computation cost of the server is reduced from $O(nm)$ to $O(n)$ in the search phase. The computation cost of the client is reduced from $O(n)$ to $O(n_a)$, where n_a is the number of documents which contain w_a. See Table 3.

Table 3. A Variant with No Add

	search	modify	delete
communication overhead	$n + O(\lambda)$	$O(\lambda)$	$O(\lambda)$
computation cost of the server	$O(n)$	$O(n)$	$O(n)$
computation cost of the client	$O(n_a)$	$O(1)$	$O(1)$

References

1. Bellovin, S., Cheswick, W.: Privacy-Enhanced Searches Using Encrypted Bloom Filters, Cryptology ePrint Archive, Report 2006/210 (2006), `http://eprint.iacr.org/`
2. Bellare, M., Desai, A., Jokipii, E., Rogaway, P.: A Concrete Security Treatment of Symmetric Encryption. In: FOCS 1997, pp. 394–403 (1997)
3. Ballard, L., Kamara, S., Monrose, F.: Achieving Efficient Conjunctive Keyword Searches over Encrypted Data. In: Qing, S., Mao, W., López, J., Wang, G. (eds.) ICICS 2005. LNCS, vol. 3783, pp. 414–426. Springer, Heidelberg (2005)
4. Byun, J.W., Lee, D.-H., Lim, J.: Efficient conjunctive keyword search on encrypted data storage system. In: Atzeni, A.S., Lioy, A. (eds.) EuroPKI 2006. LNCS, vol. 4043, pp. 184–196. Springer, Heidelberg (2006)
5. Benaloh, J., de Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 274–285. Springer, Heidelberg (1994)
6. Barić, N., Pfitzmann, B.: Collision-free accumulators and fail-stop signature schemes without trees. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 480–494. Springer, Heidelberg (1997)
7. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols, Revision 1 of ECCC Report TR01-016 (2001)
8. Canetti, R.: Universally Composable Signatures, Certification and Authentication, Cryptology ePrint Archive, Report 2003/239 (2003), `http://eprint.iacr.org/`
9. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols, Cryptology ePrint Archive, Report 2000/067 (2005), `http://eprint.iacr.org/`
10. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: ACM Conference on Computer and Communications Security, pp. 79–88 (2006)
11. Full version of the above: Cryptology ePrint Archive, Report 2006/210 (2006), `http://eprint.iacr.org/`
12. 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, Part I. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013)
13. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010)
14. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 61–76. Springer, Heidelberg (2002)
15. Chang, Y.-C., Mitzenmacher, M.: Privacy Preserving Keyword Searches on Remote Encrypted Data. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005)
16. Gennaro, R., Halevi, S., Rabin, T.: Secure hash-and-sign signatures without the random oracle. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 123–139. Springer, Heidelberg (1999)
17. Goh, E.-J.: Secure Indexes. Cryptology ePrint Archive, Report 2003/216 (2003), `http://eprint.iacr.org/`
18. Goodrich, M.T., Papamanthou, C., Tamassia, R.: On the Cost of Persistence and Authentication in Skip Lists. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 94–107. Springer, Heidelberg (2007)

19. Golle, P., Staddon, J., Waters, B.: Secure Conjunctive Keyword Search over Encrypted Data. In: Jakobsson, M., Yung, M., Zhou, J. (eds.) ACNS 2004. LNCS, vol. 3089, pp. 31–45. Springer, Heidelberg (2004)
20. Kirsch, A., Mitzenmacher, M., Wieder, U.: More Robust Hashing: Cuckoo Hashing with a Stash. SIAM J. Comput. 39(4), 1543–1561 (2009)
21. Kurosawa, K., Ohtaki, Y.: UC-Secure Searchable Symmetric Encryption. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 285–298. Springer, Heidelberg (2012)
22. Kamara, S., Papamanthou, C.: Parallel and Dynamic Searchable Symmetric Encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013)
23. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM Conference on Computer and Communications Security, pp. 965–976 (2012)
24. Merkle Tree, `http://en.wikipedia.org/wiki/Merkle~tree`
25. Song, D., Wagner, D., Perrig, A.: Practical Techniques for Searches on Encrypted Data. In: IEEE Symposium on Security and Privacy 2000, pp. 44–55 (2000)
26. Wang, P., Wang, H., Pieprzyk, J.: Keyword Field-Free Conjunctive Keyword Searches on Encrypted Data and Extension for Dynamic Groups. In: Franklin, M.K., Hui, L.C.K., Wong, D.S. (eds.) CANS 2008. LNCS, vol. 5339, pp. 178–195. Springer, Heidelberg (2008)

A Proof of Theorem 3

(1) Suppose that the real world adversary **A** does not corrupt any party in our protocol. Then it is easy to see that the client outputs the correct documents for each search keyword. Further $\mathcal{Z}$ interacts only with the client $(= P_1)$. Therefore no $\mathcal{Z}$ can distinguish the real world from the ideal world.

(2) Suppose that $\mathcal{Z}$ asks **A** to corrupt the client $(= P_1)$ in our protocol. In this case, **A** may report the communication pattern of the client to $\mathcal{Z}$. Consider an ideal world adversary **S** who runs **A** internally by playing the role of the server $(= P_2)$, forwarding all messages from $\mathcal{Z}$ to **A** and vice versa. Note that **S** can play the role of the server faithfully because it has no interaction with $\mathcal{Z}$. This means that no $\mathcal{Z}$ can distinguish the real world from the ideal world.

(3) Suppose that $\mathcal{Z}$ asks **A** to corrupt the server $(= P_2)$. In this case, our ideal world adversary **S** runs **A** internally by playing the role of the client $(= P_1)$, forwarding all messages from $\mathcal{Z}$ to **A** and vice versa.

(Store) Suppose that $\mathcal{Z}$ sends a store command to P_1. P_1 relays it to $\mathcal{F}$. $\mathcal{F}$ then sends $|D_1|, \cdots, |D_n|$ and $|\mathcal{W}|$ to **S**.

1. **S** runs the client's algorithm on input $\mathcal{D}' = \{D'_i = 0^{|D_i|} \mid i = 1, \cdots, n\}$, $\mathcal{W}' = \{1, \cdots, m\}$ and $\texttt{Index}' = \{e'_{i,j}\}$ with $e'_{i,j} = 0$ for all (i, j).
2. By doing so, **S** sends (N, g, f, k_a) and $(\mathcal{I}, \mathcal{C})$ to **A**, and keeps
$$sk = (p, q, k_e, k_0, k_1, \sigma)$$
secret, where $\mathcal{C} = (C_1, \cdots, C_n)$ and $\mathcal{I} = \{(\texttt{label}_{\sigma(i)}, \overline{\texttt{index}}_{\sigma(i)})\}$.

(Search) Suppose that $\mathcal{Z}$ sends the ith search command on a keyword $w_a \in \mathcal{W}$ to P_1. P_1 relays it to $\mathcal{F}$. $\mathcal{F}$ then sends $\texttt{List}(w_a) = \{j \mid D_j \text{ contains } w_a\}$ to **S**.

1. Let
$$e_j = \begin{cases} 1 \text{ if } j \in \texttt{List}(w_a) \\ 0 \; otherwise \end{cases} .$$
S computes $\texttt{pad}^* = \overline{\texttt{index}}_{\sigma(i)} \oplus (e_1, \cdots, e_n)$ and sends $(\texttt{label}_{\sigma(i)}, \texttt{pad}^*)$ to **A**.
2. **A** returns $(\texttt{C}'(w_a), \pi_C, \pi_I)$.
3. **S** runs the client's algorithm on input $(\texttt{C}'(w_a), \pi_C, \pi_I)$ and sk. If the client outputs `reject`, then **S** sends `reject` to $\mathcal{F}$. Otherwise **S** sends OK to $\mathcal{F}$.

(Modify) Suppose that $\mathcal{Z}$ sends a modify command (i, D_i') to P_1. Then **S** is given $|D_i'|$ by $\mathcal{F}$.

1. **S** first computes $C_i' = E_{k_e}(0^{|D_i'|})$.
2. Then **S** runs our protocol (Modify) with **A** by playing the role of the client.
3. If the client outputs `reject`, then **S** sends `reject` to $\mathcal{F}$. Otherwise **S** sends OK to $\mathcal{F}$.

(Delete) Suppose that $\mathcal{Z}$ sends a modify command i to P_1. Then **S** is given i by $\mathcal{F}$. **S** runs our protocol (Delete) with **A** by playing the role of the client. If the client outputs `reject`, then **S** sends `reject` to $\mathcal{F}$. Otherwise **S** sends OK to $\mathcal{F}$.

(Add) Suppose that $\mathcal{Z}$ sends an add command D to P_1. Then **S** is given $|D|$ by $\mathcal{F}$. **S** first computes $C_{n+1} = E_{k_e}(0^{|D|})$. **S** then runs our protocol (Add) with **A** by playing the role of the client. If the client outputs `reject`, then **S** sends `reject` to $\mathcal{F}$. Otherwise **S** sends OK to $\mathcal{F}$.

Now because `SKE` is CPA-secure, each $E_{k_e}(D)$ and $E_{k_e}(0^{|D|})$ are indistinguishable in the store phase, in the search phase, when modifying a document, and when adding a document. Further because `PRF` is a pseudo-random function, we can see that:

– The real $\mathcal{I}$ and the simulated one are indistinguishable.
– In the search phase, the real `pad` and the simulated $\texttt{pad}^*$ are indistinguishable.
– When adding a document, the real $(a_1, \cdots, a_m)$ and the simulated one are indistinguishable.

Therefore the inputs to **A** inside of **S** are indistinguishable from those in the real world. This means that inside of **S**, **A** behaves in the same way as in the real world.

We next show that the outputs of the client (which $\mathcal{Z}$ receives) in the real world are indistinguishable from those in the ideal world. Remember that **A** inside of **S** behaves in the same way as in the real world.

For a modify query (i, D_i'),

1. the client sends (i, C_i') to the server, and
2. the server returns $(H(C_i), \pi_i)$ to the client.

First suppose that **A** returns $(H(C_i), \pi_i)$ correctly.

– In the real world, the client updates A_C correctly, and outputs nothing.
– In the ideal world, **S** returns OK to $\mathcal{F}$, and $\mathcal{F}$ replaces D_i with D_i'.

Next suppose that **A** returns an invalid $(H(C_i), \pi_i)$. Then eq.(7) does not hold with overwhelming probability from Proposition 3. Hence

– In the real world, the client outputs `reject`, and $\mathcal{Z}$ receives `reject`.
– In the ideal world, **S** returns `reject` to $\mathcal{F}$, $\mathcal{F}$ sends it to P_1, and P_1 relays it to $\mathcal{Z}$.

Therefore the real world and the ideal world are indistinguishable.

Similarly, for a delete query, the real world and the ideal world are indistinguishable.

For an add query D, the client receives nothing from the server (= **A**). Hence he always updates A_C and A_I correctly, and outputs nothing.

Finally for a search query on a keyword w,

1. the client sends $(\mathtt{label}, \mathtt{pad})$ to the server, and
2. the server returns $(\mathtt{C}'(w), \pi_C, \pi_I)$ to the client, where $\mathtt{C}'(w) = \{(i, C_i) \mid D_i \text{ contains } w\}$.

First suppose that **A** returns $(\mathtt{C}'(w), \pi_C, \pi_I)$ correctly.

– In the real world, the client outputs $\mathtt{D}(w) = \{D_i \mid D_i \text{ contains } w\}$ correctly.
– In the ideal world, **S** returns OK to $\mathcal{F}$, and $\mathcal{F}$ sends $\mathtt{D}(w)$ to P_1.

Next suppose that **A** returns an invalid $(\mathtt{C}''(w), \pi_C', \pi_I')$ such that

$$(\mathtt{C}''(w), \pi_C', \pi_I') \neq (\mathtt{C}'(w), \pi_C, \pi_I).$$

We will show that eq.(6) or eq.(5) does not hold with overwhelming probability.

– (Case 1) $\mathtt{C}''(w) = \mathtt{C}'(w)$ and $(\pi_C', \pi_I') \neq (\pi_C, \pi_I)$. In this case, the client computes $\{z_j\}$ and $\{x_i\}$ correctly. Hence eq.(6) or eq.(5) does not hold clearly because $(\pi_C', \pi_I') \neq (\pi_C, \pi_I)$.
– (Case 2) $\mathtt{C}''(w) \neq \mathtt{C}'(w)$. If the client does not compute $\{z_j\}$ correctly, then we can see that eq.(6) does not hold from Proposition 3.
  Suppose that the client computes $\{z_j\}$ correctly. Then he reconstructed $(e_1, \cdots, e_n)$ and $\overline{\mathtt{index}}_a$ correctly. This means that there exist some $(i, C_i') \in \mathtt{C}''(w)$ and $(i, C_i) \in \mathtt{C}'(w)$ such that $C_i' \neq C_i$ because $\mathtt{C}''(w) \neq \mathtt{C}'(w)$. For such i, $H(i, H(C_i')) \neq H(i, H(C_i))$ because H is collision-resistant. Hence eq.(5) does not hold from Proposition 3 because $prime(H(i, H(C_i'))) \neq prime(H(i, H(C_i)))$.

Therefore in the real world, the client outputs `reject`, and $\mathcal{Z}$ receives `reject`. In the ideal world, **S** returns `reject` to $\mathcal{F}$, $\mathcal{F}$ sends it to P_1, and P_1 relays it to $\mathcal{Z}$. Consequently, we can see that $\mathcal{Z}$ cannot distinguish the real world from the ideal world. Q.E.D.