Search Result Verifiability in Multi-User Dynamic Searchable Symmetric Encryption

Masaharu Son, Takeshi Nakai and Koutarou Suzuki

September 19, 2024

Search Result Verifiability in Multi-User Dynamic Searchable Symmetric Encryption

Masaharu Son
Toyohashi University of Technology
Aichi, Japan
son.masaharu.gy@tut.jp

Takeshi Nakai
Toyohashi University of Technology
Aichi, Japan
nakai@cs.tut.ac.jp

Koutarou Suzuki
Toyohashi University of Technology
Aichi, Japan
suzuki@cs.tut.ac.jp

***Abstract*—Dynamic Searchable Symmetric Encryption (DSSE) enables a single user to retrieve and update an encrypted database stored on an external server without decryption. Multi-User DSSE (MUDSSE) enables a data owner to give access rights to multiple users, and the users perform keyword searches on the encrypted database. This paper shows a concrete construction of *verifiable* MUDSSE, which allows users to verify the correctness of their search results, for the first time. Our construction is achieved by extending the method proposed by Bost et al. (ePrint 2016) for converting a (single-user) DSSE into a verifiable one to a method applicable to MUDSSE.**

***Index Terms*—Data outsourcing, Multi-user dynamic searchable symmetric encryption, Verifiability**

I. Introduction

A. Backgrounds

Cloud services allow users to offload his/her database to an external server that holds much larger storage. Encrypting a database is a useful way to prevent information leaks when we move it to an external server. However, traditional encryption methods come with a trade-off: while they prevent the server from viewing the database's contents, they also hinder the server's ability to perform keyword searches. This trade-off means that the solution sacrifices efficiency in favor of privacy.

Searchable Symmetric Encryption (SSE) is a cryptographic protocol proposed to tackle the abovementioned problem [8], [17]. An SSE scheme enables a user to perform keyword searches on an encrypted database without decrypting it. One of the key features of SSE is its ability to allow some leakages of insignificant information, such as access and search patterns [2], [4]. This feature is not a compromise, but a strategic decision to achieve practical efficiency. SSE schemes can be classified according to the functionalities they provide:

Dynamic or Static: A *dynamic* SSE scheme allows a user to update the encrypted database after outsourcing it [2], [4], [5], [12]. In contrast, a *static* SSE schemes that do not provide such update functionality, and thus, once the user outsources a database to the server, he/she cannot modify it.

Verifiable or Unverifiable: A *verifible* SSE scheme , which considers the possibility of a server maliciously altering the search results, enables a user to verify the validity of the results [1], [13], [14], [20], [21]. In contrast, traditional (*unverifiable*) SSE schemes do not provide such functionality, and the user cannot detect errors in search results.

Single-user or Multi-user: A *single-user* SSE involves a single user and a server, which is the mainstream in researches of SSE. This setting supposes that only one user has an access right to the outsourced database. A *multi-user* SSE scheme, in contrast, involves three types of parties: a data owner, multiple users, and a server. A data owner outsources encrypted database to an external server and gives access rights of the database to users. Users retrieve the database based on the given access rights. Note that since access rights are different for each users, two users searching for the same keywords may get different search results. Specifically, a collusion-resistant multi-user SSE scheme guarantees that users cannot learn more information about the database than their access rights even if they collude [11], [15], [16], [18], [19].

Recently, Chamani et al. [6] formalized a Multi-User Dynamic SSE (MUDSSE) scheme with collusion-resistance. (To the best of our knowledge, their work is the only work to address MUDSSE with collusion resistance.) In an MUDSSE scheme, only the owner can update the database, and users can only perform keyword searches. In addition to showing an MUDSSE scheme, they claim that the proposed scheme can be extended to a verifiable MUDSSE scheme by applying a similar method based on a verifiable hash table proposed by Bost et al. [3]. However, they showed no concrete construction. Moreover, Bost et al.'s work shows a general transformation method of a verifiable dynamic SSE scheme from a (non-verifiable) dynamic SSE scheme, but it addresses only the single-user setting. Thus, it is non-trivial whether it is possible to convert an MUDSSE scheme to a verifiable one in a similar way to their method.

We remark that a searchable encryption can be constructed by using ORAM [10] or fully homomorphic encryption [9], but such constructions are inefficient.

B. Our contribution

This paper presents a concrete construction of verifiable MUDSSE for the first time. Our construction is achieved by extending Bost et al.'s method [3] to MUDSSE. In the extension, there are two main issues that need to be addressed.

This work was supported by JSPS KAKENHI Grant Number JP23K16880.

The first issue is that, since access rights differ for each user in the multi-user setting, correctness of search results also differ for each user. Our construction resolves this issue by the key idea that we make a verifiable hash table to each user. Each table is made by using an private key shared between the owner and only the corresponding user. Hence, even if the server colludes with some users, it cannot learn anything about the access rights of the other users, i.e, the scheme achieves the collusion resistance.

The second problem arises from the fact that the party updating the database, i.e., the owner, is different from the parties retrieving it, i.e., users. The verifiable hash table stored on the server needs to be updated as the DB is updated, and the local state information used for the search has to be modified after each update. Hence, as with the outsourced DB, updating and retrieving the verifiable has table must be achieved by coordinating between the owner and users. In our construction, the owner takes the responsibility for updating the hash table, while users retrieve it to perform the verification of the search results. The owner achieves local state sharing by sending the revised local state information to the user each time the table is updated. This collaborative approach ensures the validity of verification system, engaging both the owner and the users in the process.

II. Preliminaries

A. Notations

For a finite set $\mathcal{X}$, we denote by $x \xleftarrow{\$} \mathcal{X}$ the process of sampling an element x from $\mathcal{X}$ uniformly at random. $|\mathcal{X}|$ means the number of elements of $\mathcal{X}$. The empty set is denoted by $\emptyset$.

For an interactive algorithm A run between parties P_1 and P_2, $(\mathsf{out}_1; \mathsf{out}_2) \leftarrow \mathsf{A}(\mathsf{in}_1; \mathsf{in}_2)$ means that in_1 (resp. in_2) is P_1's (resp. P_2's) input. Similarly, out_1 (resp. out_2) is P_1's (resp. P_2's) output. We notate interactive algorithms involving three or more parties in the same manner.

Denote by λ security parameter. We suppose that all parties are probabilistic polynomial time (PPT) algorithms in λ. A function $v(\cdot)$ is negligible in λ if for every positive polynomial $p(\cdot)$, there exists an integer k such that for all integers $n > k$ it holds that $v(\lambda) < 1/p(\lambda)$.

Let DB be a set of target file identifiers and let W be a set of distinct keywords used in DB. Suppose that a database DB consists of keyword and file pairs, and $(w, id) \in \mathsf{DB}$ means that file $id \in D$ contains keyword $w \in W$. Then, we suppose that $|\mathsf{DB}|$ is polynomial in λ. We denote by $\mathsf{DB}(w)$ the set of file identifiers containing w and by $\mathsf{Kw}(id)$ the set of keywords in id. Let U be a set of users. For $u \in U$, we denote by $\mathsf{FileList}(u)$ a set of file identifiers to which user u has the access right, and define $\mathsf{Access} := \{\mathsf{FileList}(u)\}_{u \in U}$. Let $\mathsf{UserList}(id)$ denote a set of users who have access right to id.

B. Pseudo-Random Function

Let $\mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$ be a key generating algorithm. We say that $F : \{0,1\}^\lambda \times \{0,1\}^l \to \{0,1\}^{l'}$ is a familiy of pseudo-random functions (PRF) if for any PPT algorithm Adv, it satisfies the following property.

$$|\Pr[S \leftarrow \mathsf{Gen}(1^\lambda); \mathbf{Adv}^{F(S,\cdot)}(1^\lambda) = 1] - \Pr[\mathsf{Adv}^{R(\cdot)} = 1]| \leq v(\lambda),$$

where v is a negligible function and $R : \{0,1\}^l \to \{0,1\}^{l'}$ is a random function.

C. Multi-set Hash Function

Multi-set hash function is a variant of hash function to deal with sets as input [7].

Definition 1: Let $\mathcal{H} : \mathbb{S}^{\mathbb{Z}} \to \mathbb{F}_q$. We say a tuple of PPT algorithms $(\mathcal{H}, +_{\mathcal{H}}, -_{\mathcal{H}}, \equiv_{\mathcal{H}})$ is a collision resistant multi set hash function if for any $S \in \mathbb{S}$ it satisfies the following properties:

Comparability.
$\mathcal{H}(S) \equiv_{\mathcal{H}} \mathcal{H}(S)$

Insertion incrementality.
$\forall x \in \mathbb{S} \backslash S, \mathcal{H}(S \cup \{x\}) \equiv_{\mathcal{H}} \mathcal{H}(S) +_{\mathcal{H}} \mathcal{H}(\{x\})$

Deletion incrementality.
$\forall x \in S, \mathcal{H}(S \backslash \{x\}) \equiv_{\mathcal{H}} \mathcal{H}(S) -_{\mathcal{H}} \mathcal{H}(\{x\})$

Collision resistance.
Any PPT algorithm is a computationality hard to find two sets S_1 and S_2 such that $S_1 \neq S_2$ and $\mathcal{H}(S_1) \equiv_{\mathcal{H}} \mathcal{H}(S_2)$.

D. Verifiable Hash Table

In our proposed scheme, we use a hash table T such that for a keyword w, $T[w] = \mathcal{H}(\mathsf{DB}(w))$.

A verifiable hash table is a tuple of algorithms Θ =(VHTSetup, VHTUpdate, VHTRefresh, VHTGet, VHTVerify):

- $(K_{\mathsf{VHT}}, \mathsf{VHT}, \sigma_{\mathsf{VHT}}) \leftarrow \mathsf{VHTSetup}(T)$: It takes as input hash table T and outputs a private key K_{VHT}, verifiable hash table VHT, and state σ_{VHT}.
- $(\mathsf{VHT}, \pi) \leftarrow \mathsf{VHTUpdate}(T, \mathsf{VHT}, \gamma)$: It takes as input hash table T together with its verifiable hash table VHT, and an update operation γ. Then, it outputs the new verifiable hash table VHT and an update proof π. In this paper, the form of γ is (hkey, v), which means the value associated to hkey is overwritten with v.
- $\sigma_{\mathsf{VHT}} \leftarrow \mathsf{VHTRefresh}(K_{\mathsf{VHT}}, \sigma_{\mathsf{VHT}}, \pi, \gamma)$: It takes as input a private key K_{VHT}, state σ_{VHT}, a proof π, an update operation γ. Then it outputs (refreshed) state σ_{VHT_u}.
- $(v, \pi) \leftarrow \mathsf{VHTGet}(T, \mathsf{VHT}, \mathsf{hkey})$: It takes hash table T, verifiable hash table VHT, and hkey. It outputs the tuple (v, π) where v is the value associated to hkey in T, and a proof π.
- $y \leftarrow \mathsf{VHTVerify}(K_{\mathsf{VHT}}, \sigma_{\mathsf{VHT}}, \mathsf{hkey}, v, \pi)$: It returns $y \in \{\mathsf{ACCEPT}, \mathsf{REJECT}\}$.

VHTSetup is used to initiate a verifiable hash table. VHTUpdate is used to update the verifiable hash table when the underlying hash table is updated. VHTRefresh is used to modify the local state after running an update operation. VHTGet is used to retrieve the underlying hash table, and obtain the corresponding proof, and then VHTVerify is used to verify the retrieval result using the proof.

The verifiable hash table ensures the following two properties. (See [3] for the formal definitions of them.)

- Completeness: VHTGet returns to the value v associated to the given hkey together with a valid proof.
- Soundness: Without state σ_{VHT}, it is hard for an adversarial server to forge a valid proof, even if the server can learn a polynomial number of valid proofs.

III. (Verifiable) Multi-User Dynmanic SSE

A. MUDSSE: Multi-User Dynamic SSE

We here give the definition of MUDSSE. An MUDSSE scheme involves three types of parties: a data owner, users, and a server. It consists of four algorithms $(\mathsf{Setup}, \mathsf{Share}, \mathsf{Update}, \mathsf{Search})$ defined as follows.

- $(K, \sigma, \mathbf{EDB}) \leftarrow \mathsf{Setup}(1^\lambda, U, \mathsf{Access}, \mathsf{DB})$ is a non-interactive algorithm executed by an owner. Given a security parameter λ, a user list U, an initial access list Access, an initial database DB, it outputs a master key K, an initial state σ, and an initial encrypted database $\mathbf{EDB}$. The master key K includes a secret key $\{K_u\}_{u \in U}$ for each user.
- $(R, \sigma_u; \mathbf{EDB}) \leftarrow \mathsf{Search}(K_u, \sigma_u, w; \mathbf{EDB})$ is an interactive algorithm between the user u and the server. Given K_u, state σ_u, a keyword w from user u, and $\mathbf{EDB}$ from the owner, it outputs the search result R and updated state σ_u for the user, and updated $\mathbf{EDB}$ for the owner.
- $(\mathsf{Access}, \sigma; \mathbf{EDB}) \leftarrow \mathsf{Share}(K, u, \mathsf{Kw}(id), id, \mathsf{Access}, \sigma; \mathbf{EDB})$ is an interactive algorithm between the owner and the server. The owner inputs inputs master key K, user u, list of keywords $\mathsf{Kw}(id)$, file identifier id, σ, Access, and the server inputs $\mathbf{EDB}$. The owner gets updated state σ and Access, and the server gets updated $\mathbf{EDB}$, as output.
- $(\sigma, \mathsf{Kw}(id); \mathbf{EDB}) \leftarrow \mathsf{Update}(K, id, \mathsf{WList}, \mathsf{op}, \mathsf{Access}, \sigma; \mathbf{EDB})$ is an interactive algorithm between the owner and the server. The owner inputs master key K, file identifier id, list of keywords $\mathsf{Kw}(id)$, operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, Access, and σ, and the server inputs $\mathbf{EDB}$. The owner gets updated state σ and a list of Keywords $\mathsf{Kw}(id)$, and the server get updated $\mathbf{EDB}$, as output.

Setup is used to initiate an MUDSSE scheme. After running the algorithm, the owner distributes users' secret key for each user, and outsources $\mathbf{EDB}$ to the server. Search is used to perform keyword searches by users. Share is used to give an access right to a specified user by the owner. Update is used to add or delete keywords WList from a specified file.

B. Adaptive security

We here introduce the adaptive security of MUDSSE parameterized by leakage functions $\mathcal{L} := (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Upd}, \mathcal{L}^{Shr})$, where $\mathcal{L}^{Stp}$ corresponds to the leakage function for the setup algorithm, and likewise for the rest of them. Intuitively, each leakage function represents the information that is allowed to be leaked to the server in each operation, and we say an MUDSSE scheme is secure if it ensures that the server learn no more information than the allowed one.

The security definition follows the real/ideal simulation paradigm. (See [6] the formal description.) We consider two experiments: a real experiment ($\mathrm{REAL}_{\mathcal{A}}^{U,C,\Pi}$) in which the MUDSSE scheme is performed in the real world and an ideal experiment ($\mathrm{IDEAL}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{U,C,\Pi}$) that at most leaks information specified by a leakage function $\mathcal{L}$. In the real experiment, an adversary $\mathcal{A}$ corrupting some users $C \subsetneq U$ interacts with the algorithms of the scheme Π. In the ideal experiment, $\mathcal{A}$ interacts with a simulator that is only given information specified by $\mathcal{L}$. Then, if there we can make up a simulator such that an adversary $\mathcal{A}$ cannot distinguish between the two experiments, then Π leaks no more information than the leakage function $\mathcal{L}$.

Definition 2 ($\mathcal{L}$-adaptive security): An MUDSSE scheme Π is $\mathcal{L}$-adaptively-secure in the presence of corrupted particioants $C \subset U$ with respect to leakage function $\mathcal{L}$, iff for any PPT adversary $\mathcal{A}$ issuing a polynomial number of queries Q, there exists a stateful PPT sumulator $\mathcal{S}$ and a negligible function v such that $|\Pr[\mathrm{REAL}_{\mathcal{A}}^{U,C,\Pi} = 1] - \Pr[\mathrm{IDEAL}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{U,C,\Pi}) = 1]| \leq v(\lambda)$.

C. Verifiable MUDSSE

Verifiable MUDSSE is a variant of MUDSSE that allows users to verify the validity of their search results. We here give the definition of verifiable MUDSSE.

- $(K, \sigma, \mathbf{EDB}) \leftarrow \mathsf{VSetup}(1^\lambda, U, \mathsf{Access}, \mathsf{DB})$ is a non-interactive algorithm executed by an owner. Given a security parameter λ, a user list U, an initial access list Access, an initial database DB, it outputs a master key K, an initial state σ, and an initial encrypted database $\mathbf{EDB}$. The master key K includes a secret key $\{K_u\}_{u \in U}$ for each user.
- $(R \cup \{\mathsf{REJECT}\}, \sigma_u; \mathbf{EDB}) \leftarrow \mathsf{VSearch}(K_u, \sigma_u, w; \mathbf{EDB})$ is an interactive algorithm run between user u and the owner. Given K_u, state σ_u, search keyword w from user u, and $\mathbf{EDB}$ from the owner, it outputs the search result R or REJECT and updated state σ_u for the user, and updated $\mathbf{EDB}$ for the owner. Note that REJECT means that the user determines the search result is wrong.
- $(\sigma_u; \mathsf{Access}, \sigma; \mathbf{EDB}) \leftarrow \mathsf{VShare}(\perp; K, u, \mathsf{Kw}(id), id, \mathsf{Access}, \sigma; \mathbf{EDB})$ is an interactive algorithm run by users U, the owner, and the server. Users have not input, the owner inputs master key K, user u, list of keywords $\mathsf{Kw}(id)$, file identifier id, σ, Access, and the server inputs $\mathbf{EDB}$. The algorithm outputs a updated state σ_u for each user $u \in U$, updated state σ and an updated access list Access for the owner, and updated $\mathbf{EDB}$ for the server.
- $(\sigma_u; \sigma, \mathsf{Kw}(id); \mathbf{EDB}) \leftarrow \mathsf{VUpdate}(\perp; K, id, \mathsf{WList}, \mathsf{op}, \mathsf{Access}, \sigma; \mathbf{EDB})$ is an interactive algorithm run by users U, the owner, and the server. Users have not input, the owner inputs file identifier id, list of keywords $\mathsf{Kw}(id)$, operation $op \in \{\mathsf{add}, \mathsf{del}\}$, Access, σ, and the server

inputs $\mathbf{EDB}$. The algorithm outputs a updated state σ_u for each user $u \in U$, updated state σ and Access for the owner, and updated $\mathbf{EDB}$ for the server.

Definition 3 (Correctness): Let $R_{u,w} := \mathsf{DB}(w) \cap \mathsf{FileList}(u)$, which means the desired search result of keyword w for user u. We say a verifiable MUDSSE scheme is correct if for any user u and keyword w, VSearch fulfills the following properties:

- If $R = R_{u,w}$, the algorithm outputs R as the search result, except for negligible probability.
- Otherwise, the algorithm outputs REJECT, except for negligible probability.

Remark: In our formalization of verifiable MUDSSE, VShare and VUpdate involve users unlike the definition of (non-verifiable) MUDSSE shown in Section III-A. We require this change for technical reasons. Precisely, in our construction, each time an owner updates the verifiable hash table, he/she must share local state with users. Ideally, these communications between the owner and users should be removed. We leave the task of removing them as future work.

IV. Our Construction of Verifiable MUDSSE

This section presents our construction of VMUDSSE. Our construction follows Bost et al.'s work that show a general conversion method from (single-user) dynamic SSE into verifiable one.

A. Our Construction of Verifiable MUDSSE

Our scheme is shown in Algorithms 1–4, which is constructed based on a (non-verifiable) MUDSSE scheme Π. Suppose that Π satisfies $\mathcal{L}$-adaptively-secure for $\mathcal{L}_\Pi = (\mathcal{L}_\Pi^{Stp}, \mathcal{L}_\Pi^{Srch}, \mathcal{L}_\Pi^{Upd}, \mathcal{L}_\Pi^{Shr})$. Let $F : \{0,1\}^\lambda \times \{0,1\}^\ell \to \{0,1\}^\lambda$ be a PRF, where $\ell := |W|$.

VSetup. The algorithm is shown in Algorithm 1. The algorithm is given security parameter λ, a user list $|U|$, an initial access list Access, an initial DB as input. The algorithm runs Π.Setup algorithm that generates the owner's secret key K_Π, each user's secret key K_{Π_u}, and an encrypt database $\mathbf{EDB}$, where Π is an underlying (non-verifiable) MUDSSE scheme. Afterwards, for all $u \in U$, the algorithm makes up a hash table T_u such that its key is $\mathsf{wtag} := F(K_{T,u}, w)$ and the corresponding value is a multi-set hash value $\mathcal{H}(\{\mathsf{val}_{u,w}\}_{w \in \mathsf{DB}(w) \cap \mathsf{FileList}(u)})$. Then, it generates a verifiable hash table VHT_u of T_u for all u. At the end of the algorithm, the owner sends four keys $(K_{\mathsf{VHT}_u}, K_{\Pi,u}, K_{T,u}, K_{S,u})$ for each user u, where K_{VHT_u} is a private key of verifiable hash table, $K_{\Pi,u}$ is a private key used in Π, $K_{T,u}$ and $K_{S,u}$ are private keys of pseudo-random function F. Also, the owner sends $(\mathbf{EDB}, \{T_u, \mathsf{VHT}_u\}_{u \in U})$ to the server.

VSearch. The algorithm is shown in Algorithm 2. At the beginning, a user u obtains a search result R from the search algorithm in Π. Then, the user verifies whether the search result R is correct or not using verifiable hash table VHT_u. The user generates $\mathsf{wtag} := F(K_{T,u}, w)$, which is the key of VHT, and sends it to the server along with the user identifier u. Note that the identifier allows the server to specify the corresponding verifiable hash table. The server executes VHTGet algorithm to get value h and proof π corresponding to wtag from VHT_u. Then, the server returns (h, π) to the user u. The user u executes VHTVerify to verify if the hash table includes (wtag, h). If it outputs ACCEPT, the user compares h with $\mathcal{H}(F(K_{e,u}, R))$. If the equation holds, the user determines that R is correct. If VHTVerify returns REJECT or the equation does not hold, the user determines that R is wrong, and returns REJECT.

VUpdate. The VUpdate algorithm is presented in Algorithm 3. At the beginning, the owner updates the encrypted database and the access list by the update algorithm in Π. Then, the owner should modify the (verifiable) hash tables according to the update. Note that $\{T_u\}_{u \in \mathsf{UserList}(id)}$ are the verifiable hash tables that should be reflected the updates. If the operation of update is add, the algorithm adds $\mathcal{H}(F_{e,u}(\{id\}))$ to the values corresponding to key $\mathsf{wtag} := F(K_{T,u}, w), w \in \mathsf{Wlist}$ on the hash table for each user $u \in \mathsf{UserList}(id)$. If the operation of update is del, the algorithm subtracts $\mathcal{H}(F_{e,u}(\{id\}))$ from the values corresponding to key $\mathsf{wtag} := F(K_{T,u}, w)$ for all $w \in \mathsf{Wlist}$ on the hash table for each user $u \in \mathsf{UserList}(id)$. Afterwards, the server activates VHTUpdate algorithm to update verifiable hash table VHT_u for all $u \in \mathsf{UserList}(id)$, to updated hash tables in verifiable hash tables $\{\mathsf{VHT}_u\}_{u \in \mathsf{UserList}(id)}$. Furthermore, the owner refreshes σ_{VHT_u} by activating VHTRefresh algorithm to reflect updated information in σ_{VHT_u}. After that, the owner sends σ_{VHT_u} to users.

VShare. The VShare algorithm is presented in Algorithm 4. At the beginning, the owner updates the encrypted database by the share algorithm in Π. Then, the owner should modify the verifiable hash tables according to the update. Then, the only hash table that has to reflect the update is the VHT_u, where u is the object of the share algorithm. The algorithm adds $\mathcal{H}(F_{e,u}(\{id\}))$ to the values corresponding to key $\mathsf{wtag} := F(K_{T,u}, w)$ for all $w \in \mathsf{Kw}(id)$ on the hash table of u. Afterwards, the server activates VHTUpdate algorithm to update verifiable hash table VHT_u, to reflect updated hash table in the verifiable hash table. Furthermore, the owner refreshes σ_{VHT_u} by activating VHTRefresh algorithm to reflect the update information γ in σ_{VHT_u}. After that, the owner sends σ_{VHT_u} to the user.

B. Correctness

We here discuss correctness of our scheme. Our scheme satisfies correctness described in Definition 3.

The key values of hash tables T_u are wtag, which is uniquely determined by a keyword. For each wtag, the corresponding value is the multi-set hash of the desired search result, i.e., $h = \mathcal{H}(F(K_{e,u}, R))$. Also, in the search algorithm (Algorithm 2), a user verifies the search result R' by computing $h' := \mathcal{H}(F(K_{e,u}, R'))$, and comparing the value with $h := \mathcal{H}(F(K_{e,u}, R))$. (See line 14 of Algorithm 2.) From the comparability, the insertion incrementatlity, and the deletion incrementality of the multi-set hash, if $R = R'$, the equation $h = h'$ holds the first item of Definition 3. Note that R refers to the desired search result, and R' is the search result obtained by running the search algorithm (Σ.Search). Similarly, from the collision resistance of the multi-set hash, if $R \neq R'$, the equation does not hold except for negligible probability. Thus, our scheme fulfills the second item of Definition 3.

C. Security

Let query list L be the set of all operations of each round, and its elements are described as $(t, \mathsf{Search}, u, w)$ for a search, $(t, \mathsf{Share}, \mathsf{op}, u, id, \mathsf{WList})$ for a share and $(t, \mathsf{Update}, \mathsf{op}, id, \mathsf{WList})$ for update, where $op \in \{\mathsf{add}, \mathsf{del}\}$ and t refers to the round number. We denote by $\mathsf{qp}(u, w)$ a set of round numbers of queries in L that correspond to user u and w.

Theorem 1: Our scheme fulfills $\mathcal{L}$-adaptively-secure with the following leakage functions $\mathcal{L} := (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Upd}, \mathcal{L}^{Shr})$:

- $\mathcal{L}^{Stp}(\mathsf{DB}, U, \mathsf{Access}) = (\mathcal{L}_{\Pi}^{Stp}(\mathsf{DB}, U, \mathsf{Access}), \{|\bigcup_{id \in \mathsf{FileList}(u)} \mathsf{Kw}(id)|\}_{u \in U}, U)$,
- $\mathcal{L}^{Srch}(\mathsf{DB}, u, w) = (\mathcal{L}_{\Pi}^{Srch}(\mathsf{DB}, u, w), u, \mathsf{qp}(u, w))$
- $\mathcal{L}^{Upd}(\mathsf{DB}, op, id, \mathsf{WList}) = (\mathcal{L}_{\Pi}^{Upd}(\mathsf{DB}, op, id, \mathsf{WList}), op, \{\mathsf{qp}(u, w)\}_{w \in \mathsf{WList}}\}, \mathsf{UserList}(id), |\mathsf{WList}|)$
- $\mathcal{L}^{Shr}(\mathsf{DB}, u, id, \mathsf{Access}) = (\mathcal{L}_{\Pi}^{Shr}(\mathsf{DB}, u, id, \mathsf{Access}), u, \{\mathsf{qp}(u, w)\}_{w \in \mathsf{Kw}(id)}, |\mathsf{Kw}(id)|)$,

where $(\mathcal{L}_{\Pi}^{Stp}, \mathcal{L}_{\Pi}^{Srch}, \mathcal{L}_{\Pi}^{Upd}, \mathcal{L}_{\Pi}^{Shr})$ are leakage functions of the underlying MUDSSE scheme Π.

We defer the proof to the full version. Note that $|\bigcup_{id \in \mathsf{FileList}(u)} \mathsf{Kw}(id)|$ refers to the number of rows in hash table T_u. The above leakage functions imply that our scheme allows the server to learn the query pattern. This is due to the fact that wtag is determined uniquely from user identifier u and keyword w.

V. Conclusion

We presented a concrete construction of verifiable MUDSSE with collusion resistance for the first time. Our construction was achieved by extending the method proposed by Bost et al. [3] for transforming a (single-user) DSSE into a verifiable one to a method applicable to MUDSSE.

As a future work, the update and share algorithms should be modified to two-party protocols between the owner and the server.

References

[1] James Alderman, Christian Janson, Keith M. Martin, and Sarah Louise Renwick. Extended functionality in verifiable searchable encryption. In *Cryptography and Information Security in the Balkans*, pages 187–205. Springer International Publishing, 2016.

[2] Raphael Bost. $\Sigma o\varphi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 1143–1154. Association for Computing Machinery, 2016.

[3] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. *IACR Cryptol. ePrint Arch.*, 2016:62, 2016.

[4] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 1465–1482. Association for Computing Machinery, 2017.

[5] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21st Annual Network and Distributed System Security Symposium, NDSS*, page 23–26, 2014.

[6] Javad Ghareh Chamani, Yun Wang, Dimitrios Papadopoulos, Mingyang Zhang, and Rasool Jalili. Multi-user dynamic searchable symmetric encryption with corrupted participants. *IEEE Transactions on Dependable and Secure Computing*, 20(1):114–130, 2023.

[7] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *Advances in Cryptology - ASIACRYPT 2003*, pages 188–207, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[8] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, page 79–88. Association for Computing Machinery, 2006.

[9] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, mar 2010.

[10] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, may 1996.

[11] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 1353–1364. Association for Computing Machinery, 2016.

[12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, page 965–976. Association for Computing Machinery, 2012.

[13] Kaoru Kurosawa and Yasuhiro Ohtaki. Uc-secure searchable symmetric encryption. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, pages 285–298, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] Kaoru Kurosawa and Yasuhiro Ohtaki. How to update documents verifiably in searchable symmetric encryption. In *Cryptology and Network Security*, pages 309–328. Springer International Publishing, 2013.

[15] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Symmetric searchable encryption with sharing and unsharing. In *Computer Security*, pages 207–227, Cham, 2018. Springer International Publishing.

[16] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 157–172, Seattle, WA, 2014. USENIX Association.

[17] Dawn Xiaoding Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55, 2000.

[18] Cédric Van Rompay, Refik Molva, and Melek Önen. Secure and scalable multi-user searchable encryption. In *Proceedings of the 6th International Workshop on Security in Cloud Computing*, SCC '18, page 15–25. Association for Computing Machinery, 2018.

[19] Yun Wang and Dimitrios Papadopoulos. Multi-user collusion-resistant searchable encryption with optimal search time. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, page 252–264. Association for Computing Machinery, 2021.

[20] Dandan Yuan, Shujie Cui, and Giovanni Russello. We can make mistakes: fault-tolerant forward private verifiable dynamic searchable symmetric encryption. In *Proceedings - 7th IEEE European Symposium on Security and Privacy, EUROS&P 2022*, pages 587–605. IEEE, Institute of Electrical and Electronics Engineers, 2022.

[21] Jie Zhu, Qi Li, Cong Wang, Xingliang Yuan, Qian Wang, and Kui Ren. Enabling generic, verifiable, and secure data search in cloud services. *IEEE Transactions on Parallel and Distributed Systems*, 29(8):1721–1735, 2018.

Algorithm 1 Our Construction (Setup)

$\mathsf{VSetup}(1^\lambda, U, \mathsf{Access}, \mathsf{DB})$

Owner:
$(K_\Pi, \{K_{\Pi,u}\}_{u \in U}, \mathbf{EDB}) \leftarrow \Pi.\mathsf{Setup}(1^\lambda, U, \mathsf{Access}, \mathsf{DB})$
For all $u \in U$, initialize T_u to an empty hash table. For all $u \in U$ and $w \in W$, $\mathsf{val}_{u,w} := \emptyset$
for all $u \in U$ **do**
  $K_{T,u}, K_{S,u} \xleftarrow{\$} \{0,1\}^\lambda$
  for all $w \in W$ **do**
    $\mathsf{wtag} \leftarrow F(K_{T,u}, w)$
    $K_{e,u} \leftarrow F(K_{S,u}, w)$
    for all $id \in \mathsf{DB}(w)$ **do**
      If $id \in \mathsf{FileList}(u)$, $\mathsf{val}_{u,w} = \mathsf{val}_{u,w} \cup \{F(K_{e,u}, id)\}$
    end for
    If $\mathsf{val}_{u,w} \neq \emptyset$, $T_u[\mathsf{wtag}] \leftarrow \mathcal{H}(\mathsf{val}_{u,w})$
  end for
end for
$(K_{\mathsf{VHT}_u}, \mathsf{VHT}_u, \sigma_{\mathsf{VHT}_u}) \leftarrow \mathsf{VHTSetup}(T_u)$
Set $K_u = (K_{\mathsf{VHT}_u}, K_{\Pi,u}, K_{T,u}, K_{S,u})$ for all $u \in U$
Set $K = (K_\Pi, \{K_u\}_{u \in U})$
Send $(\mathbf{EDB}, \{T_u, \mathsf{VHT}_u\}_{u \in U})$ to the server
Send $(K_u, \sigma_{\mathsf{VHT}_u})$ to user u for all $u \in U$

Algorithm 2 Our Construction (Search)

$\mathsf{VSearch}(K_u, \sigma_u, w; \mathbf{EDB})$

$(R, \sigma_u; \mathbf{EDB}) \leftarrow \Pi.\mathsf{Search}(K_{\Pi,u}, \sigma_u, w; \mathbf{EDB})$

User u:
$\mathsf{wtag} \leftarrow F(K_{T,u}, w)$
$K_{e,u} \leftarrow F(K_{S,u}, w)$
Send (wtag, u) to the server

Server:
$(h, \pi) \leftarrow \mathsf{VHTGet}(T_u, \mathsf{VHT}_u, \mathsf{wtag})$
Send (h, π) to the user u

User u:
if $\mathsf{ACCEPT} \leftarrow \mathsf{VHTVerify}(K_{\mathsf{VHT}_u}, \sigma_{\mathsf{VHT}_u}, \mathsf{wtag}, h, \pi)$
  if $h \equiv_{\mathcal{H}} \mathcal{H}(F(K_{e,u}, R))$
    return R
else
  return REJECT

Algorithm 3 Our Construction (Update)

$\mathsf{VUpdate}(K, id, \mathsf{WList}, \mathsf{op}, \mathsf{Access}, \sigma; \mathbf{EDB})$

$(\sigma, \mathsf{Kw}(id); \mathbf{EDB}) \leftarrow \Pi.\mathsf{Update}(K_\Pi, id, \mathsf{WList}, op, \mathsf{Access}, \sigma; \mathbf{EDB})$
for all $u \in \mathsf{UserList}(id)$ **do**
  for all $w \in \mathsf{WList}$ **do**
    Owner:
    $\mathsf{wtag} \leftarrow F(K_{T,u}, w)$
    $K_{e,u} \leftarrow F(K_{S,u}, w)$
    $h' \leftarrow \mathcal{H}(F_{K_{e,u}}(\{id\}))$
    Send $(\mathsf{wtag}, h', \mathsf{op}, u)$ to the server

    Server:
    $h \leftarrow T_u[\mathsf{wtag}]$
    if $\mathsf{op} = \mathsf{add}$
      $h'' \leftarrow h +_{\mathcal{H}} h'$
    if $\mathsf{op} = \mathsf{del}$
      $h'' \leftarrow h -_{\mathcal{H}} h'$
    Let $\gamma = (\mathsf{wtag}, h'')$, which means the overwriting of $T_u[\mathsf{wtag}]$ by h''
    $(\mathsf{VHT}_u, \pi) \leftarrow \mathsf{VHTUpdate}(T_u, \mathsf{VHT}_u, \gamma)$
    Send (π, h, h'', γ) to the owner

    Owner:
    $\sigma_{\mathsf{VHT}_u} \leftarrow \mathsf{VHTRefresh}(K_\sigma, \sigma_{\mathsf{VHT}_u}, \pi, \gamma)$
    Send σ_{VHT_u} to the user
  end for
end for

Algorithm 4 Our Construction (Share)

$\mathsf{VShare}(K, u, \mathsf{Kw}(id), \mathsf{Access}, \sigma_u; \mathbf{EDB})$

$(\mathsf{Access}, \sigma_u; \mathbf{EDB})$
  $\leftarrow \Pi.\mathsf{Share}(K_\Pi, u, \mathsf{Kw}(id), id, \mathsf{Access}, \sigma_u; \mathbf{EDB})$
for all $w \in \mathsf{Kw}(id)$ **do**
  Owner:
  $\mathsf{wtag} \leftarrow F(K_{T,u}, w)$
  $K_{e,u} \leftarrow F(K_{S,u}, w)$
  $h' \leftarrow \mathcal{H}(F_{K_{e,u}}(\{id\}))$
  Send (wtag, h') to the server

  Server:
  $h \leftarrow T_u[\mathsf{wtag}]$
  $h'' \leftarrow h +_{\mathcal{H}} h'$
  Let $\gamma = (\mathsf{wtag}, h'')$, which means the overwriting of $T_u[\mathsf{wtag}]$ by h''
  $(\mathsf{VHT}_u, \pi) \leftarrow \mathsf{VHTUpdate}(T_u, \mathsf{VHT}_u, \gamma)$
  Send (π, h, h'', γ) to the owner

  Owner:
  $\sigma_{\mathsf{VHT}_u} \leftarrow \mathsf{VHTRefresh}(\sigma_{\mathsf{VHT}_u}, \gamma)$
  Send σ_{VHT_u} to the user.
end for