Enabling Generic, Verifiable, and Secure Data Search in Cloud Services

Jie Zhu, Qi Li, *Senior Member, IEEE*, Cong Wang, *Senior Member, IEEE*, Xingliang Yuan, *Member, IEEE*, Qian Wang, *Member, IEEE*, Kui Ren, *Fellow, IEEE*

Abstract—Searchable Symmetric Encryption (SSE) has been widely studied in cloud storage, which allows cloud services to directly search over encrypted data. Most SSE schemes only work with honest-but-curious cloud services that do not deviate from the prescribed protocols. However, this assumption does not always hold in practice due to the untrusted nature in storage outsourcing. To alleviate the issue, there have been studies on Verifiable Searchable Symmetric Encryption (VSSE), which functions against malicious cloud services by enabling results verification. But to our best knowledge, existing VSSE schemes exhibit very limited applicability, such as only supporting static database, demanding specific SSE constructions, or only working in the single-user model. In this paper, we propose GSSE, the first generic verifiable SSE scheme in the single-owner multiple-user model, which provides verifiability for any SSE schemes and further supports data updates. To generically support result verification, we first decouple the proof index in GSSE from SSE. We then leverage Merkle Patricia Tree (MPT) and Incremental Hash to build the proof index with data update support. We also develop a timestamp-chain for data freshness maintenance across multiple users. Rigorous analysis and experimental evaluations show that GSSE is secure and introduces small overhead for result verification.

1 Introduction

Cloud storage allows users to retrieve and share their data conveniently with well understood benefits, such as on-demand access, reduced data maintenance cost, and service elasticity [1]–[7]. Meanwhile, cloud storage also brings serious data privacy issues, i.e., the disclosure of private information. In order to ensure data privacy without losing data usability, a cryptographic notion named searchable symmetric encryption (SSE), (e.g., [8]–[12], to just list a few), has been proposed. By using SSE, users can encrypt their data before uploading to cloud services, and cloud services can directly operate and search over encrypted data, which ensures data privacy.

However, most existing SSE schemes [9]–[11] are built based on the assumption that cloud services are honest but curious, which means cloud services will follow the protocol but intend to derive users' information from their search queries. Unfortunately, this assumption does not always hold in practice, since cloud services may be subject to external attacks, internal misconfiguration errors, software bugs, and even insider threats [7], [13]. All these factors may cause the cloud services to deviate from the prescribed protocol and operate beyond the honest-but-curious model. Exemplary consequences might be cloud services executing a fraction of search operations or omitting some files in search results.

In order to address this issue, a large amount of studies [1], [4], [5], [20]–[22] have been conducted to ensure data integrity against a malicious cloud server. Also, verifiable SSE schemes [3], [13]–[19] have been developed to ensure data integrity in SSE. Unfortunately, these schemes either support verification on only static database [14], [15], [18], [19], or cannot prevent cloud services from deliberately returning an empty result to evade result verification [3], [16], [17]. Specifically, previous schemes that are built on Merkle Hash Tree [3], RSA accumulator [16], or Message Authenticated Code (MAC) [17] are not able to return any search result when there does not exist any document matching the query keywords [13]. To prevent the server from returning an empty result maliciously, the user should maintain all keywords of the data set locally. Recently, Ogata et al. [19] addressed the issue by maintaining keywords with a cuckoo hash table. Unfortunately, the scheme cannot enable verification under data updates. Further, most verifiable SSE schemes [3], [13]–[19] only enable verifiability for the single-user model, which we refer to as the two-party model. However, in practice, service providers such as public cloud normally enable data sharing among the data owner and multiple data users in a three-party model, where data owner and user are not the same entity. Table 1 compares various existing verifiable SSE schemes. To our best knowledge, none of the existing verifiable SSE schemes can explicitly allow users to verify their search results in the three-party model.

In this paper, we propose `GSSE`, a generic dynamic verifiable SSE framework to ensure search result integrity and freshness across multiple users. It can be applied to any SSE schemes, including but not limited to those in [10], [11], [17], etc., to provide search results verification for data users. In addition, it supports data updates, a highly desir-

- *J. Zhu and Q. Li are with Graduate School at Shenzhen, Tsinghua University, Shenzhen, China, and Dept. of Computer Science, Tsinghua University, Beijing, China. E-mail: j-zhu15@mails.tsinghua.edu.cn, qi.li@sz.tsinghua.edu.cn.*
- *C. Wang is with Dept. of Computer Science, City University of Hong Kong, Hong Kong. E-mail: congwang@cityu.edu.hk.*
- *X. Yuan is with Faculty of Information Technology, Monash University, Australia. E-mail: xingliang.yuan@monash.edu.*
- *Q. Wang is with School of Computer Science, Wuhan University, Wuhan, China. E-mail: qianwang@whu.edu.cn.*
- *K. Ren is with Institute of Cyber Security Research, Zhejiang University, China. E-mail: kuiren@zju.edu.cn.*

TABLE 1: Comparison with existing typical verifiable SSE schemes.

	Dynamism	Three-party[1]	Freshness Verify[2]	Integrity Verify[3]	Prove Efficiency[4]	Generality[5]
KPR11 [3]	✓	×	✓	×	$O(\|W\|)$	✓
KO12 [14]	×	×	-	×	$O(n)$	×
CG12 [15]	×	×	-	✓	$O(log(\|W\|))$	×
KO13 [16]	✓	×	✓	×	$O(n)$	×
SPS14 [17]	✓	×	✓	×	$min\{\alpha + log(N), rlog^3(N)\}$	×
CYGZR15 [18]	×	×	-	×	$O(\|W\|) + O(r)$	×
BFP16 [13]	✓	×	✓	✓	$O(r)$	✓
OK16 [19]	×	×	-	✓	$O(r)$	✓
GSSE	✓	✓	✓	✓	$O(log(\|W\|))$	✓

[1] Three-party means whether the scheme supports search result verification for an SSE scheme with three parties, i.e., data owners, servers, and users.
[2] Note that, '×' represents the requirements which are not implemented, while '-' means the requirements which are not required. Specifically, the static verifiable SSE schemes do not have the problem of data freshness attacks, and thus the existing schemes [14], [15], [18], [19] do not require data freshness verification.
[3] We consider various data integrity attacks, especially the attacks that servers can intentionally returns an empty result to evade search result verification.
[4] The prove efficiency refers to the cost of operations for search result verification. For some selected non-generic schemes [14]–[18], their prove efficiency is equivalent to their encrypted search efficiency. Here, n indicates the number of total files, $|W|$ means the number of all keywords, r means the number of files which contain the specific keyword, α means the number of times this keyword was historically added to the collection [17], and N means the total number of document and keyword pairs.
[5] A generic VSSE scheme means that the verifiable design can provide result verification for any SSE schemes, while a non-generic scheme only works for a particular SSE construction.

able advantage demanded by many modern cloud storage applications, where data update happens frequently.

GSSE addresses two challenges in verifying search results of SSE. The first challenge is how to design an efficient yet generic proof index which not only supports data integrity verification but also supports data updating. GSSE builds and maintains such a proof index by leveraging the fully dynamic and balanced Merkle Patricia Tree (MPT) [23] and Incremental Hash [24]. With these two prelimitives, we store encrypted keywords and their corresponding documents in the proof index such that the root of the MPT becomes an accumulator of the data, which can be treated as a witness of data integrity. Meanwhile, GSSE designs a verification mechanism based on the proof index to ensure the authenticity of search results. Different from the previous solutions [3], [16], [17], our scheme requires the server to return a proof to the users regardless of whether the keyword exists or not, such that the users can detect whether the cloud services deliberately omit all files and returning an empty result to evade result verification. More specially, GSSE does not require the users to maintain a large set of keywords, while easily verifying the integrity of the search results with the proof.

The second challenge is how to ensure data freshness by preventing the root from being replayed in the context of data updates. In the previous two-party model, data ower can recalculate the root after each update, but in the three-party model, data users cannot easily detect a data update from the data owner, unless data owner sends the latest root to all users after each data update. But doing so would bring in significant, if not impractical, online communication burden to the data owner. In order to solve this problem, we develop a timestamp-chain based verification mechanism for GSSE. This mechanism constructs a timestamp-chain based authenticator which includes the root of the MPT. It allows users to obtain an authenticator from cloud services on demand and easily ensure the freshness of the root while not incurring significant computation and communication overhead. In summary, our contributions are three-fold:

1) We propose the first generic verifiable SSE framework, i.e., GSSE, in the single-owner multiple-user model, which provides verifiability for any existing SSE schemes and further supports data updates.
2) We develop verification mechanisms for GSSE such that it can ensure both the freshness and integrity of search results across multiple users and data owners. Rigorous analysis formally shows the security strength of GSSE.
3) Through comprehensive experimental results, we show that GSSE only introduces small extra overhead for result verification, compared to existing searchable encryption schemes.

2 Related Work

Secure Cloud Storage Scheme. Verifiable cloud storage services have been extensively studied, e.g., Proof of Data Possession (PDP) [2], [20]–[22] and Proof of Retrievability (POR) [1], [5], [25]. These schemes mainly focused on verifying the integrity of data stored in cloud services and enable restoring data blocks if they are corrupted. However, they did not ensure the integrity of search results, which is the focus of VSSE. Authenticated data structures are used by a set of searching algorithms to verify the integrity of data blocks stored on an untrusted server. Several schemes have been proposed, e.g., Merkle Tree [26], authenticated hash table [27], and authenticated skip list [28], [29]. Merkle Tree is the most common structure used to verify data integrity. However, Merkle Tree cannot flexible support data update. Moreover, the current verification scheme [3] built upon Merkle Tree did not store keyword information in its intermediate node and thus it is not suitable for keyword related searches. An authenticated hash table enabled by the RSA accumulator can be used to verify search results as well. Unfortunate, it has low efficiency in searching and update operations. For example, the search delay of the authenticated hash table is in millisecond level, while that of GSSE is in microsecond level. Skip list used a multilayer linked list to improve its search efficiency, but the storage overhead is much higher than a tree structure if the keyword information is required in the search path.

Verifiable Searchable Symmetric Encryption. The CS2 scheme [3] enabled users to verify the search result by using dynamic search authenticators, but their scheme cannot prevent the attacks that the server maliciously replies an empty result. Recently Kurosawa et al. [14], [16], [19] proposed a few verifiable SSE schemes. However, their schemes either have low search efficiency, or do not support verification upon file update. Kurosawa et al. [14] required linear search in SSE and did not support dynamic file update. Their extension [16] achieved dynamic updating but the search complexity was beyond linear time. Recently, Ogata et al. [19] presented a generic verifiable scheme. It transforms any SSE scheme to a *no-dictionary* verifiable SSE scheme that did not require the users to keep the keyword set. However, it was still a static approach, which shared the similar shortcoming with [15] [18]. Although the verifiable scheme proposed by Stefanov et al. [17] achieved verifiability by leveraging message authenticated code, it cannot easily detect the data integrity attacks when the server intentionally returned an empty result. Bost et al. [13] presented a generic verifiable dynamic SSE scheme and combined it with the SSE scheme proposed by Stefanov et al. [17]. Yet, their scheme required two round communications for result verification and did not enable verification in the setting of multiple users. Our GSSE scheme is a generic verifiable SSE scheme that can work with three-party model, which can be more readily deployed in practice. In particular, it enables search result verification under file update with only one round of communication.

Verifiable Public Key Encryption with Keyword Search. The first verifiable attribute-based keyword search (VABKS) was proposed by Zheng et al. [30]. Similar to the existing SSE schemes above, VABKS only focused on search based on static encrypted data. Liu et.al [31] proposed a more efficient construction based on VABKS, and Sun et.al [7] also provided a verifiable scheme VCKS that support conjunctive keyword search. However, due to the limitations of asymmetric encryption schemes, both of the above schemes require an additional trusted authority.

Multi-User Searchable Encryption. A few of non-verifiable multi-user schemes have been proposed [9], [32]–[34]. Curtmola et al. [9] first proposed a multi-user SSE scheme based on broadcast encryption. Yang et al. [32] proposed a multi-user searchable encryption scheme by leveraging a bilinear map. However, the search delay of the scheme is proportional to the size of the database, which is not suitable for large-scale databases. Jarecki et al. [33] designed a multi-user scheme by using Oblivious Cross-Tags (OXT) protocol. However, their scheme required frequent communication between data owners and the users, which incured unnecessary communication overheads. Recently, Sun et al. [34] proposed a non-interactive multi-user searchable encryption schemes that reduced the interactions between data owner and users. However, the scheme did not support search under data update.

3 Problem Statement

In this section, we first formally define our problem and then present our design goals. We also review preliminaries used in this paper.

3.1 Threat Model

We assume that the data owner is trusted and the data users authorized by the data owner are also trusted[1]. We consider cloud services performing searchable symmetric encryption (SSE) to be untrusted, which means 1) cloud services intends to derive some sensitive information from the encrypted data and the queries; 2) cloud services may deviate from the prescribed protocols and mount a data freshness attack or a data integrity attack to save its computation or communication cost. The definitions of the data freshness attack and the data integrity attack are presented as follow:

Definition 1 (**Data Freshness Attacks**). *A data freshness attack in SSE is that a malicious server (or an attacker) attempts to return the historical version of the search result, not the most recently updated version. Formally, let* $\Delta_{n-1} = \{\delta_1, \delta_2, \cdots, \delta_{n-1}\}$ *denote the historical version of the dataset and* δ_n *is the latest version. However, the search result returned by the server is retrieved from* δ_i *where* $1 \leq i \leq n-1$.

Definition 2 (**Data Integrity Attacks**). *A data integrity attack in SSE is that a malicious server (or an attacker) attempts to tamper with the search result to prevent authenticated users from accessing the complete and correct search result. Formally, let* τ *be the search token of the SSE scheme, and* δ_i *be the dataset, where* $1 \leq i \leq n$, *the corresponding search result should be* $\mathcal{F}(\delta_i, \tau)$, *but the result returned by the server is* $\mathcal{G}(\delta_i, \tau)$, *where* $\mathcal{G}(\delta_i, \tau) \neq \mathcal{F}(\delta_i, \tau)$.

3.2 Design Goal

In this paper, we aim to design a generic verifiable SSE scheme that enables verifiable searches on the three-party model. In particular, the scheme should satisfy the following privacy and efficiency requirements:

1) **Confidentiality:** The confidentiality of data and keywords is the most important privacy requirements in SSE. It ensures that users' plaintext data and keywords cannot be revealed by any unauthorized parties, and an adversary cannot learn any useful information about files and keywords through the proof index and update tokens used in GSSE.
2) **Verifiability:** A verifiable SSE scheme should be able to verify the freshness and integrity of the search results for users.
3) **Efficiency:** A verifiable SSE scheme should achieve sublinear computational complexity, e.g. logarithmic

1. Please refer to Section 8 for details on how we can enforce such assumption in practice with multi-user access control techniques.

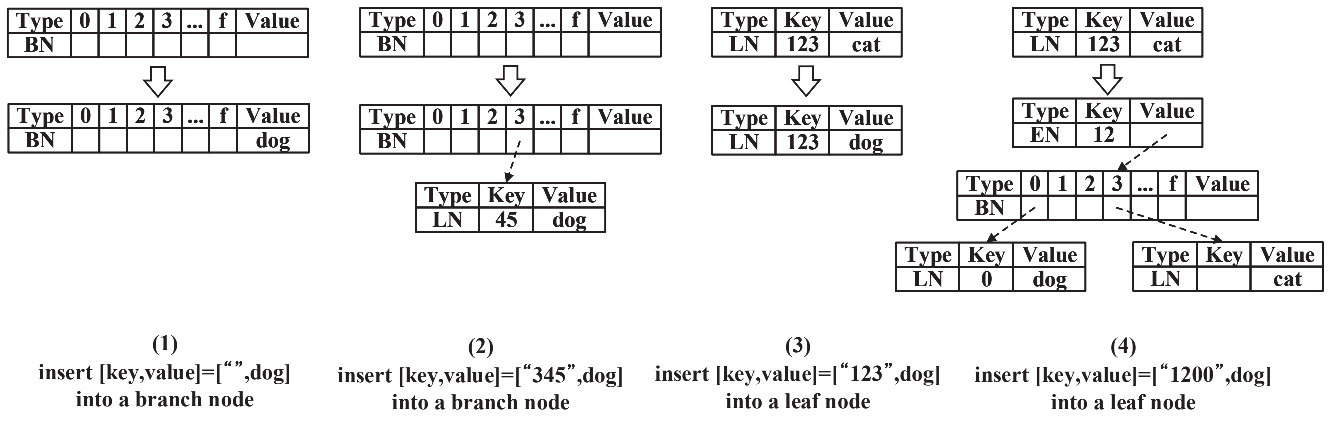


Fig. 1: The Merkle Patricia Tree

$O(log(|W|))$, where $|W|$ is the number of keywords, even with file update. Note that, the computational complexity only refers to the cost of searching operations for verification, which does not include the complexity of the searching operations in the existing SSE schemes.

This paper aims to provide result verification for any SSE schemes, including but not limited to [10], [11], [17]. Therefore, we treat an existing SSE scheme as a black box such that our proposed scheme can be applied to these SSE schemes for result verification.

3.3 Preliminaries

Merkle Patricia Tree. The Merkle Patricia Tree (MPT) is first proposed in Ethereum [23], [35], which combines the Trie Tree and the Merkle Tree for data update efficiency. There are three kinds of nodes in an MPT to achieve the goal. Leaf Nodes(LN) represents [key,value] pairs. Extension Nodes(EN) represent [key,value] pairs where keys are the public prefixes and their values are the hashes of the next nodes. The Branch Nodes (BN) are used to store possible branches when the prefixes of the keywords differ, which is presented with 17 elements. Among the 17 elements, the first 16 elements represent the 16 possible hex characters in a key and the last element stores a value if a key in a [key,value] pair matches the node. Fig. 1 shows insertion operations of a Merkle Patricia Tree (MPT) with the following four cases. First, to insert a [key,value] pair into a branch node, there are two possible cases. If the current key is empty, we can directly insert the value into the 17th bucket of the branch node. Otherwise, the unmatched key and value will be stored in a leaf node. Second, if we want to insert a [key,value] pair into a leaf node, there are also two possible cases. If the current key matches, we should modify the value of the leaf node directly. Otherwise, we should find the common prefix as the key of a newly created extension node. Meanwhile, we create a new branch node, and the original leaf node and the inserting [key,value] pair will be inserted as child node of the branch node. Note that, each node of the MPT is represented by its hash and is encoded using Recursive Length Prefix (RLP) code that is mainly used to encode arbitrarily binary data [36], which ensures the cryptographically security of the search operations. The root hash in MPT becomes a fingerprint of the entire tree and is computed based on all hashes of nodes below. Therefore, any modification in a node would incur recomputation of the root hash. Note that, the MPT is fully deterministic, meaning that an MPT with the same [key,value] pairs is exactly the same regardless of the order of insertion, which is different from the Merkle Tree.

Incremental Hash. Incremental hash was proposed by Bellare et al. [24] and was used by existing SSE schemes, e.g., CS2 [3]. An incremental hash function is a collision-resistant function $IH : \{0,1\}^* \rightarrow \{0,1\}^l$, with which the addition or the subtraction operation of two random strings on the IH does not produce a collision. For example, assuming F is a file collection that contains the keyword k. After a new file f is inserted to F, the file collection becomes F' (i.e., $F+f$), which means the new file f is a slight change according to F. Therefore, an incremental hash function can be used to quickly compute the corresponding collision-resist hash value after a file change. More detailed descriptions can be found in [3].

Secure Searchable Encryption. Searchable Encryption was first proposed by Song et al. [8], their solution allows a user to outsource its encrypted data to cloud services, and meanwhile retaining the ability to search over it. Normally, searchable encryption has been divided into two categories, i.e., Searchable Symmetric Encryption(SSE) and Public Key Encryption with keyword search(PKE). The most classical SSE scheme was proposed by Curtmola et al. in [9]. They defined privacy against passive adversaries (i.e., honest but curious servers) and developed their scheme by using an inverted index. There exist various SSE schemes with different secure searching functionalities. For example, dynamic SSE schemes [10], [11], [17] allow a user to update his dataset and ranked keyword search scheme [37] that allow a user to retrieve ranked search results from the server. The most famous PKE scheme was proposed by Boneh et al. [38] with the bilinear map. Normally, the efficiency of the PKE schemes are much lower than the SSE schemes.

4 Overview of GSSE

In this section, we present an overview of our GSSE scheme. The major notations used in this paper are shown in Table 2.

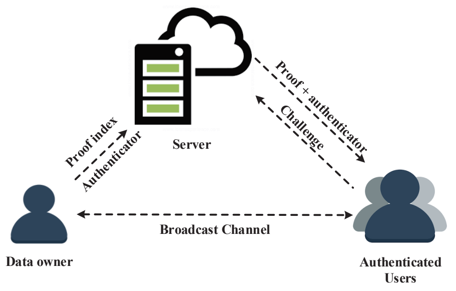


Fig. 2: System architecture of GSSE on the three-party model.

TABLE 2: Notations

Notation	Meaning
$\mathcal{W}$	keyword set
$\|W\|$	size of the keyword set
w_i	keyword, where $i \in \{1, \cdots, \|W\|\}$
$\mathcal{D}$	plaintext of the document set
D_{w_i}	plaintext of a document set containing w_i
$\mathcal{C}$	ciphertext of the document set
C_{w_i}	ciphertext of a document set containing w_i
f	plaintext of a document
c	ciphertext of a document
W_f	keyword set of document f
τ	the search token (challenge)
λ	the proof index.
π	the authenticator.

4.1 System architechture

Fig. 2 illustrates the system architecture of GSSE. It consists of three parties: *data owners*, who provide the encrypted proof index corresponding to their data and authenticators to cloud services; the *untrusted server*, which provides storage and search services; a set of *authenticated users*, who challenge the cloud services for verification of search results retrieved from the SSE scheme.

4.2 System model

We aim to develop a verifiable SSE scheme, i.e., GSSE, that allows the index used for search result verification to be separated from the one used for the SSE operations. Therefore, GSSE is decoupled from the existing SSE schemes. In particular, data owner will builds an encrypted index based on the Merkle Patricia Tree (MPT) and upload it to cloud services, which enables data users to verify the integrity of search results. Meanwhile, data owner will also upload a timestamp-chain based on the root of MPT to ensure data freshness across multiple users. GSSE is defined as follows.

Definition 3 (GSSE **Scheme).** *In a* GSSE *scheme, there are three parties, i.e., data owners, authenticated users and an untrusted server. A data owner provides a proof index and an authenticator to the untrusted server such that it allows the server to provide a proof of the search result and authenticators for the authenticated users to ensure the integrity and freshness of the SSE search results. A* GSSE *scheme is a collection of seven polynomial-time algorithms, where*

- *$KGen(1^k) \rightarrow \{K_1, K_2, K_3, (ssk, spk)\}$: is a probabilistic algorithm run by the data owner. It takes as input a security parameter, and outputs the secret keys K_1, K_2, K_3 and a random signing keypair (ssk, spk).*
- *$Init(K_1, K_2, K_3, ssk, \mathcal{D}) \rightarrow \{\lambda, \pi\}$: is an algorithms run by the data owner which takes as input the symmetric keys K_1, K_2, K_3, the signing secret key ssk, and the document set $\mathcal{D}$, and outputs the proof index λ and the anthenticator π. The data owner stores the proof λ locally and meanwhile sends λ and π to the server.*
- *$PreUpdate(K_1, K_2, K_3, ssk, f) \rightarrow \{\tau_u, \pi\}$: is an algorithm run by the data owner. It takes as input the symmetric keys K_1, K_2, K_3, the signing secret key ssk, and a file f to be updated, and outputs the update tokens τ_u and the authenticator π. The data owner sends τ_u and π to the server.*
- *$Update(\lambda, \tau_u) \rightarrow \{\lambda'\}$: is an algorithm run by the server. It takes as input the proof index λ and the update tokens τ_u, and outputs the new proof index λ';*
- *$Challenge(K_1, w) \rightarrow \{\tau_w\}$: is a deterministic algorithm run by the user. It takes as input a symmetric key K_1 and a specific keyword w, and outputs a challenge τ_w corresponding to w. The user sends the challenge τ_w to the server.*
- *$Prove(\lambda, \tau_w, t_q) \rightarrow \{\rho, \pi_q^t, \pi_c\}$: is an algorithm run by the server. It takes as input the proof index λ, the challenge τ_w, and the query time t_q, it outputs the proof ρ and authenticators π_q^t,π_c. The server sends ρ and authenticators π_q^t,π_c to the requested user.*
- *$Verify(K_1, K_2, K_3, spk, C_w, \rho, \pi_q^t, \pi_c, \tau_w) \rightarrow \{b\}$: is an algorithm run by the data user which takes as input symmetric keys K_1, K_2, K_3, the public key spk, the SSE search result C_w, the proof ρ, authenticators π_q^t, π_c and the challenge token τ_w, it outputs a bit b represent an accept or reject result. This algorithm consists of two sub-algorithms, the Check algorithm and the Generate algorithm, which can be written as $Check(K_3, spk, \pi_q^t, \pi_c) \rightarrow \{b\}$ and $Generate(K_1, K_2, K_3, C_w, \rho, \tau_w, \pi_q^t) \rightarrow \{b\}$.*

5 GSSE Construction

In this section, we present our GSSE construction, by starting with algorithms to build and update our proof index, and then detailed algorithms of our verification mechanism.

5.1 Building Proof Index

Algorithm 1 shows the pseudo-code of building the proof index and the authenticator (i.e., the *Init* algorithm defined in Definition 3). It builds a proof index with MPT structure based on the document set $\mathcal{D}$, and the inverted index Δ computed from $\mathcal{D}$, where an inverted index refers to the index that indicates the documents containing a specific keyword. For every keyword w_i in the inverted list Δ, we compute the key-value pairs which will be stored in our proof index, i.e., the MPT, where the key is the token of the distinct keyword w_i and the value is the incremental hash

Algorithm 1 *Init*

```
Input:
    $K_1, K_2, K_3$: the symmetric keys; $ssk$: the secret key for signing; $\mathcal{D}$: the document set; $F, G : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^*$ the pseudo-random functions; $IH : \{0,1\}^* \rightarrow \{0,1\}^k$ the incremental hash functions; $H : \{0,1\}^* \rightarrow \{0,1\}^k$ the hash function.
Output:
    $\lambda$: the proof index established using the MPT; $\pi$: the authenticator
1: for each $w_i \in \Delta$, where $\Delta$ is the inverted index which consists of $< w_i, D_{w_i} >$ pairs do
2:     $\tau_{w_i} = F_{K_1}(w_i)$
3:     $V_{w_i} = \sum_{f_i \in D_{w_i}} IH(G_{K_2}(f_i))$
4:     $\lambda = \lambda.Insert(\tau_{w_i}, V_{w_i})$
5: end for
6: Generate authenticator $\pi$ as Equation (1) with symmetric key $K_3$ and secret key $ssk$.[2]
7: return $\{\lambda, \pi\}$
```

Algorithm 2 Verify

```
Input: $K_1, K_2, K_3$: symmetric keys; $spk$: the public key for verifying signature; $C_w$: the search results; $\rho$: the proof of the search results; $\tau_w$: the challenge made by the user; $\pi_q^t$: the authenticator received in the query time $t$; $\pi_c$: the authenticator received in the checkpoint;
Output: $b \in \{0,1\}$, if $b = 1$, accept; otherwise, reject.
1: $b \leftarrow Check(K_3, spk, \pi_q^t, \pi_c)$
2: return $b \leftarrow b \;\; \&\& \;\; Generate(K_1, K_2, K_3, C_w, \rho, \tau_w, \pi_q^t)$
```

of all the documents which contain the keyword. Note that, the key stores on the path of the tree and the value stores on the corresponding leaf node.

The *Update* algorithm (see Definition 3) updates the proof index on the server and supports three operations, i.e., the *add*, *delete* and *edit* operations. Here, the *edit* operation is equivalent to adding a new file after deleting an old file. We briefly describe the algorithm here. First, for *add* operations, update tokens τ_u is split into the $\{\tau_{w_i}, G_{K_2}(f)\}$ pairs, where τ_{w_i} is the token of a specific keyword extracted from the file f and $G_{K_2}(f)$ is a pseudo-random string of f. We locate the corresponding leaf nodes based on its tokens and add the value $IH(G_{K_2}(f))$ to the existing node value. A new leaf node will be created if a token does not have a corresponding leaf node. The *delete* operation is similar to the *add* operation. We locate the leaf node and subtract $IH(G_{K_2}(f))$ from the value of the leaf node. Note that the *PreUpdate* algorithm performed by the data owner provides the tokens for the *Update* algorithm conducted on the server.

5.2 Verifying Search Results

Algorithm 2 shows the search result verification algorithm performed by data users (i.e., *Verify* algorithm shown in Definition 3). Firstly, it checks the correctness of the authenticator by the *Check* algorithm. Here, an authenticator is used to ensure the freshness of the root. If the authenticator is not replayed by the server, which means the root is fresh, then we use the *Generate* algorithm to verify search results by leveraging the root hash value extracted from the authenticator and the proof retrieved from the server. Finally, according to the results of *Check* and *Generate*, the algorithm can determine the freshness and integrity of the search results. In the later subsections, we will elaborate the *Check* and *Generate* algorithms.

5.3 Verifying Authenticators

In order to prevent cloud services from replaying previous authenticators and ensure the freshness of the root, we maintain a timestamp-chain for authenticators, such that users can trace authenticators in the chain and identify if the root is fresh. Here, the timestamp-chain scheme is different from the timestamp mechanism used in SSE [17]. Their schemes can only prevent servers from constructing data freshness attacks under the two-party model when the user holds the update information. Unfortunately, it may not be able to detect data freshness attack in our concerned three-party model. We show the details of our scheme below.

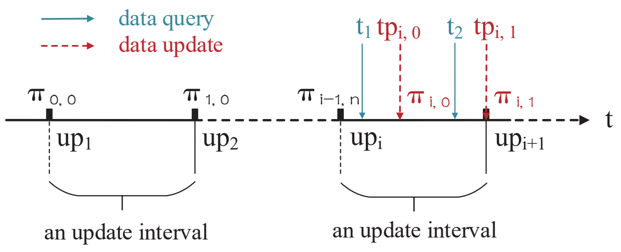


Fig. 3: An illustration of the timestamp-chain mechanism

Firstly, we will show how the data owner creates the authenticator by leveraging the timestamp-chain. In the very beginning, a data owner sets an interval for authenticator update[3], and then the fixed update time points are set to be $\{up_1, up_2, \cdots, up_i, \cdots, up_m\}$ (see Fig. 3). Here, we use Network Time Protocol (NTP) [39], [40] run on cloud servers to synchronize the clocks among the data owners and the data users during their interactions with the servers. The clock synchronization accuracy can reach a few milliseconds or even tens of microseconds [41]–[43], and thus the accuracy is enough for verification in GSSE. Note that, a malicious server can possibly fake a clock, but it cannot fake the timestamp-chain. If the server faked the clock, the timestamp will not allow a server to bypass the verification performed by users. The authenticator is uploaded to the server periodically at the update time point when there is no data update in an update interval. Otherwise, the data owner will additionally upload the authenticators along with the updating data.

Intuitively, in order to prevent the authenticator from being replayed, we can simply set the authenticator π as a concatenation of timestamp tp and the root of the MPT, i.e., the proof index, encrypt it by using a symmetric key K_3, and then sign it with the secret key ssk. If the proof index

3. In the performance evaluation section, i.e., Section 7, we will show the relationship between the update interval and the delays of detecting data freshness attacks.

Algorithm 3 Check

```
Input: $K_3$: the symmetric key; $spk$: the public key for verifying signature; $\pi_q^t$: the authenticator received in the query time $t$; $\pi_c$: the authenticator received in the checkpoint.
Output: $b \in \{0,1\}$, if $b = 1$, the Check algorithm succeeds, otherwise, it fails.
 1: let $\pi_q^t = \{\alpha_q^t, Sig_q^t\}$ and $\pi_c = \{\alpha_c, Sig_c\}$
 2: if $\alpha_q^t \neq (Sig_q^t)_{spk} || \alpha_c \neq (Sig_c)_{spk}$ then
 3:     return $b = 0$
 4: end if
 5: $(rt_q^t, tp_q^t, \alpha) \leftarrow Dec_{K_3}(\alpha_q^t)$
 6: if $tp_q^t$ is not before the previous update time point then
 7:     let $\alpha_k = \alpha_c$
 8:     for $\alpha_k \neq \emptyset$ do
 9:         $(rt_k, tp_k, \alpha_{k-1}) \leftarrow Dec_{K_3}(\alpha_k)$
10:         if $tp_k < t$ then
11:             break
12:         end if
13:         let $\alpha_k = \alpha_{k-1}$
14:     end for
15:     if $\alpha_k = \alpha_q^t || \alpha_k = \emptyset$ then
16:         return $b = 1$
17:     else
18:         return $b = 0$
19:     end if
20: else
21:     return $b = 0$
22: end if
```

is not updated during an update interval, the data owner only needs to update the timestamp at next update time point. If the document set of the data owner is modified within an update interval, which means the root of the proof index has been updated, then the data owner will calculate a new authenticator by using the latest root and the current timestamp and upload it to cloud services again. In this setting, when a data user generates a challenge, the server should send the latest authenticator to the data user. The data user can recover the root and the timestamp by decrypting the authenticator. If the timestamp is beyond the valid time, i.e., before the latest update time point, then the server is considered as malicious. This mechanism ensures that the server cannot mount a data freshness attack by using the data before the latest update time point.

However, cloud services may still be able to replay the authenticator between the latest update time point and the current query time. Specifically, if there is one or more data updates happened after the latest update time point, then the server can cheat the data user by sending any authenticator uploaded after the latest update time point and then mounts a data freshness attack within the latest update interval. Therefore, we develop a timestamp-chain mechanism to detect those cheating behaviors. We modify the structure of the authenticator by chaining the value of the previous authenticator into the newly generated authenticator according to Equation (1). Note that, it will generate a new timestamp-chain of a new update interval, while the timestamp-chain ends at the beginning of the next update interval. In other words, the authenticators in each update interval are chained together, e.g., $\pi_{i,0}, \pi_{i,1}$ (see Fig. 3), but the authenticators are irrelevant in the two different update intervals. Here, the last authenticator in each update interval is uploaded at the next update time point. In this setting, the server needs to provide an authenticator at the query time and meanwhile an authenticator at the checkpoint, where the checkpoint is referred as the next update time point closest to the user's query time t, e.g., up_{i+1} is the checkpoint in the update interval $(up_i, up_{i+1}]$.

$$\begin{cases} \pi_{i,0} = (\alpha_{i,0}, \mathsf{Sig}_{ssk}(\alpha_{i,0})), & up_i < tp_{i,0} \le up_{i+1} \\ \alpha_{i,0} = Enc_{K_3}(rt_{i,0}||tp_{i,0}) & \\ \dots & \\ \pi_{i,j} = (\alpha_{i,j}, \mathsf{Sig}_{ssk}(\alpha_{i,j})), & tp_{i,j-1} < tp_{i,j} \le up_{i+1} \\ \alpha_{i,j} = Enc_{K_3}(rt_{i,j}||tp_{i,j}||\alpha_{i,j-1}) & \\ \dots & \\ \pi_{i,n} = (\alpha_{i,n}, \mathsf{Sig}_{ssk}(\alpha_{i,n})), & tp_{i,n} = up_{i+1} \\ \alpha_{i,n} = Enc_{K_3}(rt_{i,n}||tp_{i,n}||\alpha_{i,n-1}) & \end{cases} \quad (1)$$

Here i represents the i-th update interval and j represents the j-th authenticator in the interval.

Let us consider the following cases (shown in Fig. 3) when a data user initiates a query at different time points: (i) the first case is that the query occurs at t_1, where $t_1 < tp_{i,0}$, the server can only send $\pi_{i-1,n}$ to the data user; (ii) the second case is that the query occurs at t_2 after the data update event at $tp_{i,0}$, and the authenticator that server sends to the user is $\pi_{i,0}$; (iii) the last case is that the query is generated at t_2, and the authenticator sent by the server is $\pi_{i-1,n}$. In the last case, a data freshness attack occurs, but it will be detected at the checkpoint up_{i+1}. The data user will obtain the last authenticator $\pi_{i,1}$ from the server at the checkpoint to verify whether the data obtained at the query time has been replayed or not.

Algorithm 3 shows the pseudo-code of the *Check* algorithm that is executed by a data user and verifies whether the authenticator has been replayed. Let π_q^t denote the authenticator received at the query time t and π_c denote the authenticator received at the checkpoint, which is used to deduce the previous authenticators during the latest update interval. First, we need to verify the signature of π_q^t and π_c by using the public key spk of the data owner. We check the authenticator π_q^t received at the query time is not generated before the previous update time point by using α_q^t extract from π_q^t. Then, we decrypt the previous $rt_k||tp_k||\alpha_{k-1}$ concatenation by using α_k until it finds the first concatenation with timestamp $tp_k < t$ or $\alpha_k = \emptyset$. We compare α_k with α_q^t and $\emptyset$. If it is not equal to either of them, a data freshness attack is detected. Otherwise, α_q^t is considered correct. Now we use the three cases above to explain the algorithm. In the first case, $\pi_{i,1}$ and $\pi_{i,0}$ are received and $\alpha_{i,1}$ and $\alpha_{i,0}$ are extracted. We can find the field of α in the concatenation is $\emptyset$ after decrypting $\alpha_{i,0}$. Therefore, the *Check* algorithm outputs $b = 1$ and the authenticator $\pi_{i-1,n}$ received in the query time is considered correct. In the second case, $\alpha_{i,0}$ is also decrypted by $\alpha_{i,1}$ and

Algorithm 4 Prove

```
Input: $\lambda$: the proof index maintained by server; $\tau_{w_i}$: the challenge made by an authenticated user; $t_q$: the query time of user.
Output: $\rho$: the proof of the SSE search result; $\pi_q^t, \pi_c$:the authenticators;
 1: Find the search path $\sigma = (n_0, \cdots, n_i, \cdots, n_m) \leftarrow \lambda.Search(\tau_{w_i})$, where $n_i \in \{EN, BN, LN\}$, $n_0$ is the root node.
 2: if $t_{w_i}$ exist then
 3:   for $i = m - 1$ to $0$ do
 4:     if $n_i = BN$ then
 5:       $\rho = \rho \cup C_{n_i}$ where $C_{n_i}$ includes several key-value pairs that are not on the search path and the key only which is on the search path $\sigma$
 6:     else if $n_i = EN$ then
 7:       $\rho = \rho \cup C_{n_i}$ where $C_{n_i}$ is the key which is on the search path $\sigma$
 8:     else
 9:       $\rho = \rho \cup C_{n_i}$ where $C_{n_i}$ is the key-value pair of node $n_i$
10:     end if
11:   end for
12: else
13:   for $i = m$ to $0$ do
14:     Repeat steps 4-8
15:   end for
16: end if
17: Find the latest authenticator $\pi_q^t$ according to the query time $t_q$ and the authenticator $\pi_c$ at the checkpoint.
18: return $\rho, \pi_q^t, \pi_c$
```

the timestamp of $\alpha_{i,0}$ is less than t_2. We can find that $\alpha_{i,0}$ and $\alpha_q^{t_2}$ are equal. Hence $\alpha_q^{t_2}$ is considered correct, i.e., $\pi_q^{t_2}$ is correct. However, in the last case, we will detect a data freshness attack due to the mismatch between the correct authenticator $\pi_{i,0}$ and the received one $\pi_q^{t_2}$, i.e., $\pi_{i-1,n}$.

Remark. The update interval can be controlled by the data owner according to its update frequency. Normally, if data is frequently updated, the update interval can be set to a shorter period so that the length of the authenticator will decrease and the verification delays will be shorter. However, it will incur more communication overheads. In our experiments (see Section 7), we will show that the verification delays and the bandwidth consumption for updating authenticators are acceptable.

5.4 Verifying Proofs

A user can start using the fresh root to verify the integrity of the search results after confirming that the user has obtained the correct authenticator at the query time. In order to allow data users to generate the root of the proof index to verify search results, servers need to present proof which is generated by the *Prove* algorithm. The *Prove* algorithm is performed by the server according to proof index λ , the challenge τ_{w_i} (that is received from a user and corresponds to a specific keyword w_i) and the query time t. Here, we consider both cases that the keyword is in the presence or is absence in the path of MPT. The server has to provide

Algorithm 5 Generate

```
Input: $K_1, K_2, K_3$: the symmetric keys; $C_w$: the search result; $\rho$: the proof of the search result; $\tau_w$: the challenge made by the user himself; $\pi_q^t$: the root received at the query time.
Output: $b \in \{0, 1\}$, if $b = 1$, the Generate algorithm succeeds, otherwise, it fails.
 1: Compute $\{remain\_key\}$ = String.match ($\tau_{w_i}$, keys in $\rho$)
 2: if $C_w = \emptyset$ && $remain\_key = \emptyset$ then
 3:   Calculated the root $rt$ according to $\rho$ from the bottom to root.
 4: else if $C_w \neq \emptyset$ && $remain\_key \neq \emptyset$ then
 5:   Compute $\varphi = \sum_{f \in D_w} IH(G_{K_2}(f_i))$, where $D_w$ is the plaintext of $C_w$
 6:   Compute $LN = Compute(\varphi, remain\_key)$
 7:   Calculated the root $rt$ according to $LN$ and $\rho$ from the bottom to the root.
 8: else
 9:   return 0
10: end if
11: $(rt_q^t, tp_q^t, \pi) \leftarrow Dec_{K_3}(\alpha_q^t)$, where $alpha_q^t$ is extract from $\pi_q^t$;
12: if $rt = rt_q^t$ then
13:   return 1
14: else
15:   return 0
16: end if
```

a proof if the keyword exists or a proof of absence if the keyword does not exist. The absence proof prevents the server from intentionally returning an empty result.

Algorithm 4 shows the pseudo-code of generating proofs for verification (see *Prove* algorithm defined in Definition 3). First, the server searches the proof index according to the submitted token and find the corresponding search path σ. We need to consider two cases here. If the token exists, the server needs to return the keys of each node in the search path from the bottom to the root, excluding the leaf node itself. Note that, for a branch node, we also need to return the key-value pairs that are not in the search path. However, if the token does not exist, the server also needs to return the keys of each node in the search path from the node where the search terminates to the root. Note that, we need to provide the value of the node where the search terminates. The former case is the normal one when the keyword exists, and the proof returned by the server allows the user to verify the integrity of the search results. In the latter case, the server needs to return the absence proof according to the algorithm since the proof enables the user to ensure the absence of the keyword. If the server does not follow the algorithm and returns an invalid proof, the users can detect the behaviors and the server will be treated as malicious.

After receiving the proof from the server, the data user needs to generate the tree root, which is performed by the *Generate* algorithm (see in Definition 3). The pseudo-code is shown in Algorithm 5. It first compares the challenge τ_w with the keys in ρ. If the keys in ρ is not the prefix of τ_w, $remain\_key$ is set to $\emptyset$. Otherwise, $remain\_key$ stores the remaining bits of τ_w. If both the search result and

$remain\_key$ are $\emptyset$, we can generate the tree root rt according to the proof ρ. If both the search result and the $remain\_key$ is not $\emptyset$, we need to calculate the corresponding leaf node according to the search result and the $remain\_key$, and then generate the tree root rt by using the calculated leaf node and the proof ρ. If it's neigher of the above two cases, the server is considered malicious, the server is considered malicious. Finally, we compare the calculated root rt with the correct one rt_q^t to verify the correctness of the root rt. If they are not equal, it means the server has tampered with either the proof ρ or the search result, and thus the verification fails.

5.5 An Illustrative Example

We use an example shown in Fig. 4 to exemplify the algorithms operating the proof index. We assume that initially there are four documents, i.e., $\{f_1, f_2, f_3, f_4\}$, which consist of four different keywords $\{w_1, w_2, w_3, w_4\}$ presented in the inverted list (see the left part of Fig. 4). Keyword w_1 is contained in all documents. Keyword w_2 is only contained in document f_2. keywords w_3 and w_4 are contained in $\{f_1, f_2, f_3\}$ and $\{f_1, f_2, f_4\}$ respectively. The corresponding tokens and values of keywords are also given in the inverted list. Initially, we build the proof index by inserting the key-value pairs into MPT. For an update operation, e.g., adding a new file f_5 that contains w_2 and w_5, the update tokens are split into ['a5432', $IH(G_{K_2}(f_5))$] and ['a5fab', $IH(G_{K_2}(f_5))$]. For the token 'a5432' that already exists, we only need to add $IH(G_{K_2}(f_5))$ to the original node value $IH(G_{K_2}(f_2))$. For the new token 'a5fab', we need to create a new node and assign the value $IH(G_{K_2}(f_5))$ to it. Note that any change to the node will trigger a change of the hash value in the root node.

Suppose a user wants to search the keyword w_2 and submits the corresponding token 'a5432' which already exists in the proof index. The search path of this token is {BN1,EN1,BN2,LN3}, and the proof ρ produced by the *Prove* algorithm should be $[C_{n_2}, C_{n_1}, C_{n_0}]$ (see Fig. 4). After receiving the proof ρ from the server, the data user runs the *Generate* algorithm to check the integrity of the search result. For simplicity, here, we assume that all root hash values in this example are verified by the *Check* algorithm. First, the $remain\_key$ '32' is calculated by string matching. Specifically, 'a','5','4' can be found in $C_{n_0}, C_{n_1}, C_{n_2}$, so the $remain\_key$ is '32'. Then the first shaded area in ρ is calculated based on the $remain\_key$ and the search results of the SSE scheme. Namely, for keyword w_2, the search result of the SSE scheme should be file f_2 and f_5. The user needs to recompute the value of LN3 by binding the $remain\_key$ '32' with the sum of $IH(G_{K_2}(f_2))$ and $IG(G_{K_2}(f_5))$. If the server does not cheat, the value should be equal to the value of LN3. After retrieving the value of the first shaded part, we can generate the root hash according to the proof ρ. If there is an attack, e.g., the server only returns the file f_2 that is the result before the update of file f_5, the rebuilt root hash value will also be the root hash before the update. Then the *Generate* algorithm will fail.

Now suppose the user submits a token that does not exist in the proof index, e.g., token 'a5433'. The search path in the tree terminates in the leaf node LN3, which is the same to the search path of 'a5432'. The proof ρ returned by the server should be $[C_{n_3}, C_{n_2}, C_{n_1}, C_{n_0}]$ (see Fig. 4). The user first confirms that the first bit 'a' of the token is present in C_{n_0}, and then confirms that '5' and '4' are in C_{n_1} and C_{n_2}, respectively. It is obvious that '33' does not exist in C_{n_3}, thus $remain\_key$ is set to $\emptyset$, which indicates the keyword does not exist. Then, the user generates the root hash based on the proof by using *Generate* algorithm and compares it with the root extracted from the authnticators. Note that, any small changes in the proof ρ will affect the value of the final root hash. Therefore, users can easily detect if the server tampered with the proof ρ, and then ensure the integrity of the search results.

6 Security Analysis

In this section, we give a rigorous security analysis of our GSSE scheme. We plan to demonstrate the security of GSSE on two aspects, i.e., confidentiality and verifiability. Confidentiality means an adversary cannot learn any useful information about files and keywords through the proof index and update tokens used in GSSE, while verifiability means that result verification will not output an accept when the search result received from cloud services is incorrect or incomplete. First, we adopt the **Real**/**Ideal** simulation in [3] to prove confidentiality of GSSE.

Definition 4 (GSSE **confidentiality**)**.** *Let the* GSSE *scheme be a dynamic verifiable scheme based on the searchable symmetric encryption and consider the following probabilistic experiments, where $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator, and $\mathcal{L}$ are stateful leakage algorithms:*

$\mathbf{Real}_{\mathcal{A}}(k)$: *a challenger runs $KGen(1^k)$ to generate symmetric keys K_1, K_2, K_3. The adversary $\mathcal{A}$ chooses a document set $\mathcal{D}$ for the challenger to create a proof index λ and an authenticator π via $\{\lambda, \pi\} \leftarrow Init(K_1, K_2, K_3, \mathcal{D})$, and makes a polynomial number of adaptive queries $q = \{w, f\}$. For each query q, $\mathcal{A}$ receives from the challenger a challenge token τ_w such that $\tau_w \leftarrow Challenge(K_1, w)$, an update token and the authenticator (τ_u, π) such that $(\tau_u, \pi) \leftarrow PreUpdate(K_1, K_2, K_3, f)$. Finally, $\mathcal{A}$ returns a bit b.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$: *The adversary $\mathcal{A}$ chooses a document set $\mathcal{D}$. Given $\mathcal{L}(\mathcal{D})$, the simulator $\mathcal{S}$ generates and sends proof index $\tilde{\lambda}$ and the authenticator $\tilde{\pi}$ to $\mathcal{A}$. The adversary $\mathcal{A}$ makes a polynomial number of adaptive queries $q = \{w, f\}$. For each query q, $\mathcal{S}$ returns the appropriate token τ and the authenticator π. Finally, $\mathcal{A}$ returns a bit b.*

We say that SSE is $\mathcal{L}$-confidential if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that

$$|Pr[\mathbf{Real}_{\mathcal{A}}(k) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k) = 1]| \leq negl(k).$$

Before proving the confidentiality, we formalize the view of the adversary as follows: $\mathcal{L}(\mathcal{D}) = (|\lambda|, |\pi|, \{\tau\}_q, \{\sigma\})$. Here $|\lambda|$ is the size of the proof index indicated by the number of leaf nodes. π is the length of the authenticator. $\{\tau\}_q$ are q tokens which are adaptively generated. $\{\sigma\}$ are the search paths in the proof index, e.g., all the tokens correspond to the set of keywords $\mathcal{W}$. Then we have the following theorem.

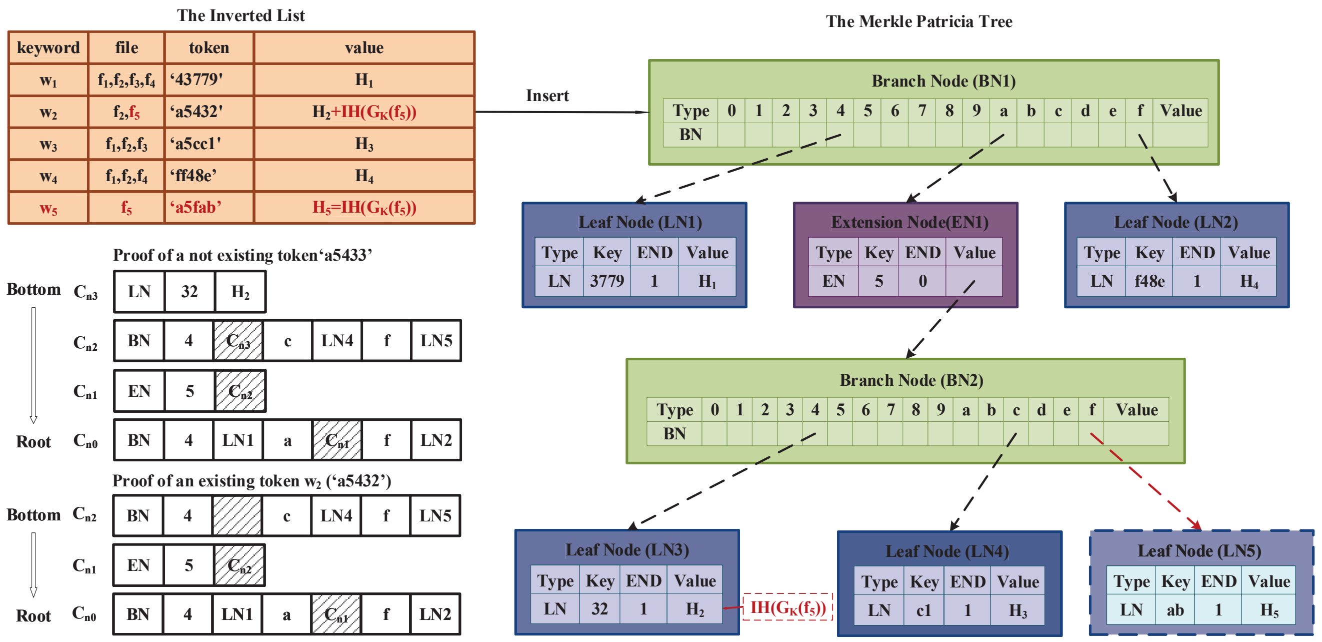

The Inverted List
keyword | file | token | value
w1 | f1,f2,f3,f4 | '43779' | H1
w2 | f2,f5 | 'a5432' | H2+IH(GK(f5))
w3 | f1,f2,f3 | 'a5cc1' | H3
w4 | f1,f2,f4 | 'ff48e' | H4
w5 | f5 | 'a5fab' | H5=IH(GK(f5))
Insert
The Merkle Patricia Tree
Branch Node (BN1)
Leaf Node (LN1)
Extension Node(EN1)
Leaf Node (LN2)
Branch Node (BN2)
Leaf Node (LN3)
Leaf Node (LN4)
Leaf Node (LN5)
IH(GK(f5))
Proof of a not existing token'a5433'
Proof of an existing token w2 ('a5432')
Bottom
Root

Fig. 4: An illustrative example

Theorem 1. *The* GSSE *scheme is $\mathcal{L}$-confidential if F and G are pseudo-random functions.*

Proof. We show that there exist a polynomial-time simulator $\mathcal{S}$ such that for all probabilistic polynomial time (PPT) adversaries $\mathcal{A}$, the output between the real game $\mathbf{Real}_{\mathcal{A}}(k)$ and a simulation game $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ is computationally indistinguishable.

First of all, given $\mathcal{L}(\mathcal{D}) = (|\lambda|, |\pi|, \{\tau\}_q, \{\sigma\})$, $\mathcal{S}$ simulates the proof index $\tilde{\lambda}$ by choosing $|\lambda|$ random key-value strings and inserting into MPT. Meanwhile, $\mathcal{S}$ chooses a random string $\tilde{\pi}$ with length $|\pi|$. Recall that each key-value pair in MPT is encrypted by the pseudo-random function F and G, and the confidentiality of the authenticator is essentially ensured by the underlying cipher. Therefore, $\mathcal{A}$ cannot differentiate $(\tilde{\lambda}, \tilde{\pi})$ from (λ, π). Now let $\mathcal{S}$ simulates challenge tokens. For the first token τ_w, if it matches one search path in $\{\sigma\}$, then $\mathcal{S}$ chooses a random path in $\tilde{\lambda}$ as the challenge token $\tilde{\tau_w}$ and returns it to $\mathcal{A}$. Otherwise, $\mathcal{S}$ chooses a random string which is not in the search path of $\tilde{\lambda}$. Thus, $\mathcal{A}$ cannot differentiate the real token from the simulated token. For the subsequent tokens, if w has appeared before, then the challenge token $\tilde{\tau_w}$ is the same to the previous one or follows the same way of simulating the first token. In either case, the challenge token $\tilde{\tau_w}$ is returned to $\mathcal{A}$ accordingly. Similarly, when $\mathcal{A}$ simulates an update token, the update token is set to $\tilde{\tau_u} = (\tilde{\tau_{w_1}}, \cdots, \tilde{\tau_{w_{|W_f|}}}, \tilde{\tau_r})$ and the authenticator $\tilde{\pi}$ is set as a random string with the same length as π. For each $\tilde{\tau_{w_1}}$, $\mathcal{A}$ chooses the random string as the same way in simulating the challenge tokens. Since all tokens in the $\mathbf{Real}_{\mathcal{A}}(k)$ game was encrypted by the pseudo-random function F, the adversary $\mathcal{A}$ cannot differentiate the simulated tokens from the real tokens. Therefore, we can conclude that the outputs of $\mathbf{Real}_{\mathcal{A}}(k)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ are indistinguishable. □

The verifiability of the GSSE scheme means that the scheme can verify the freshness and integrity of the search results, i.e., prevent the data freshness attack and data integrity attack defined by Definition 1 and Definition 2. Here, we adopt a game-based security definition to prove the verifiability of GSSE.

Definition 5 (GSSE **verifiability).** *Let the* GSSE *scheme be a dynamic verifiable scheme based on the searchable symmetric encryption and consider the following probabilistic experiments, where $\mathcal{A}$ is a stateful adversary:*
$\mathbf{Vrf}_{\mathcal{A}}(k)$*:*

1. *the challenger runs $KGen(1^k)$ to generate symmetric keys K_1, K_2, K_3.*
2. *the adversary $\mathcal{A}$ chooses a document set $\mathcal{D}$ for the challenger.*
3. *the challenger creates a proof index λ and an authenticator π via $\{\lambda, \pi\} \leftarrow Init(K_1, K_2, K_3, \mathcal{D})$,*
4. *given $\{\lambda, \pi\}$ and oracle access to $Challenge(K_1, w)$ and $PreUpdate(K_1, K_2, K_3, f)$, the adversary $\mathcal{A}$ outputs a keyword token τ_w, a sequence of files C' such that $C' \neq C_w$, the authenticators π'_q, π'_c and a proof ρ'.*
5. *the challenger computes $b := Verify(K_1, K_2, K_3, C_w, \rho, \pi'_q, \pi'_c, \tau_w)$.*
6. *the output of the experiment is the bit b.*

We say that GSSE *is verifiable if for all PPT adversaries $\mathcal{A}$,*

$$Pr[\mathbf{Vrf}_{\mathcal{A}}(k) = 1] \leq negl(k).$$

Theorem 2. *The* GSSE *scheme is verifiable if the hash function H and the incremental hash function IH are collision-resistant and G is pseudo-random.*

Proof. Considering the situation where a search result $\tilde{D_w}$ returned by the server is different from the correct answer D_w but the *Verify* algorithm accepts the search result $\tilde{D_w}$. In order to ensure the GSSE scheme is verifiable, we only need to prove verifiability of *Check* and *Generate* algorithms.

First, for the *Check* algorithm, since the authenticator π is encrypted by the data owner, its unforgeability is guaranteed by the underlying AES ciphers and digital signature. Anyone without the secret signing key ssk and symmetric key K_3 cannot generate the authenticator that can be authenticated by the data user. Second, for the *Generate* algorithm, there are two possible scenarios to output two collision root hash. The first is that $\widetilde{D_w}$ and D_w induce a collision of the incremental hash function IH. The other is that the collision occurs in the path when computing the root hash of the proof index. However, the probability that a hash function produces a collision is less than a negligible value, so the verifiability of the *Generate* algorithm is guaranteed. Therefore, the GSSE scheme is indeed verifiable. □

7 Performance Evaluation

7.1 Experiment Setup

In order to demonstrate the feasibility of GSSE, we have implemented it by using Crypto++ 5.6.5. The prototype is written by about 2200 lines of code. We use 128-bit AES-CBC to encrypt the authenticators and sign it with RSA signature. We implement two random-oracles with HMAC-SHA256 and the hash function is an implementation of SHA3-256 and the incremental hash function is MuHash. Our experiments were performed by using a machine with single thread on an Intel Core i5 2.5GHz processor with 4G RAM. We used the Enron email dataset [44] in our experiments. The used part of the dataset [44] is between "allen-p" and "kaminski-v". We extract document-keyword pairs from the dataset and construct our plaintext inverted index by using a python script. Note that the delays of extracting keywords from files are not included in our evaluation, since keyword extraction is independent with GSSE. We first measure the overheads of the algorithms proposed and then compare GSSE with a well-known SSE scheme [11] to demonstrate the small extra overhead introduced by result verification.

7.2 Experimental Results

First, we measure the delays of the *Init* algorithm as shown in Fig. 5, which include the building of the proof index and the authenticator. All the delays and the subsequent measurements are the average results with ten runs of experiments. Note that the cost of building the authenticator is negligible. The delays of generating the proof index are proportional to the size of the document-keyword pairs, since GSSE performs the same number of insertions to the number of the document-keyword pairs. Overall, the initialization consumes around 25 seconds where the documents include four million keywords, which is acceptable.

The update delays are decided by the size of the database that is measured by the number of keywords. Strictly speaking, the delays are directly related to the number of the layers in MPT. In order to show the relationship between the update delays and the database size, we use various numbers of keywords to measure the delays. Since the number of keywords varies from each file, we use throughput to measure the number of keyword-document pairs that can be updated per second (see Fig. 6). We observe that the throughput of adding and deletion operations are almost the same. The throughput decreases when the size of the database grows. They can support 110,000 updates per second with one million keywords database. Similarly, we observe that the bandwidth overhead incurred by update token is decided by the number of keywords contained in the file. Each update token takes about 32 bytes, which is acceptable as well.

As shown in Fig. 7, the server can perform about 43,000 prove operations per second even when the size of the database is one million keywords, which indicates the server can simultaneously support 43,000 concurrent queries submitted by users. Note that this experiment only measures the cost of generating the proofs, not including the waiting time for the authenticator in the checkpoint. The communication overhead incurred by proof delivery is only a few kilobytes, which is decided by the number of layers in MPT, and gradually increases as the database grows (see Fig. 7 and Fig. 8).

We measure the storage cost MPT as shown in Fig. 9. If we use a database with 1,000,000 keywords, the storage overhead is about 82MB. Compared with the size of the original dataset itself, 590 MB, the overhead is relatively small. Note that, if a data owner stores various media types of data set (e.g., images or music) with fewer keywords or attributes, the storage overhead of MPT will further reduce to be practically negligible, compared with the size of the data set itself.

The performance of *Generate* algorithm performed by data users is presented in Fig. 10. We observe that the measured delays of generating root are all within 0.1 milliseconds and are acceptable.

In Fig. 11, we evaluate the verification delays in data users. Note that an entire verification delay includes the delay of waiting for a checkpoint and the delay of executing the *Check* and the *Generate* algorithms. Since the execution delay of the *Generate* algorithm is relatively stable, around 0.1 milliseconds, we do not plot it in Fig. 11. Here, η is the update frequency of the data owner. We assume that the time that a user initiates a query is uniformly distributed during an update interval, and then the user's waiting delays are also uniformly distributed. Therefore, the expected delay is half of the update interval and the verification delays are dominated by the waiting delays. The execution delay of the *Check* algorithm is negligible and is proportional to the update interval, which is mainly incurred by verifying the signature and decrypting authenticators. Kindly note that in above measurement, we do not take into account the network transmission and propagation delays, as they vary in different specific network contexts and do not reflect the essential extra cost directly introduced by our verification design. We do, however, report the communication overhead in terms of the message size, as shown in Fig. 12. In a later experiment, we will also show that we can set an update interval so as to make a trade-off between verification delays and communication overhead.

Fig. 12 shows the bandwidth costs for authenticator update. Here, the size of the first authenticator in each update interval is around 112 bytes, which includes 32 bytes of the root of MPT, 8 bytes of the timestamp, an 8 bytes

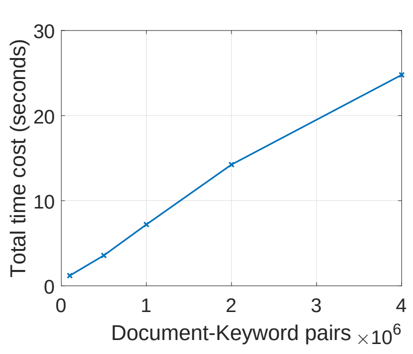


Fig. 5: *Init* delays

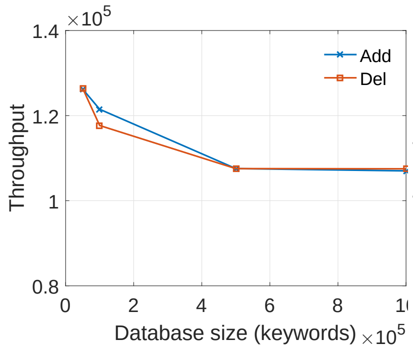


Fig. 6: *Update* throughput

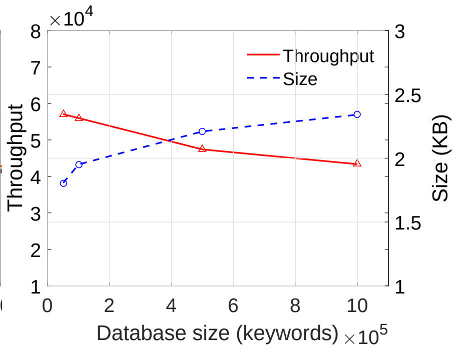


Fig. 7: *Prove* cost

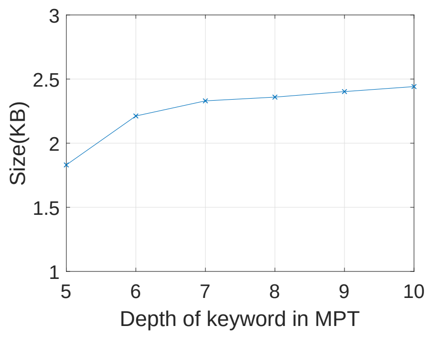


Fig. 8: Proof cost of MPT

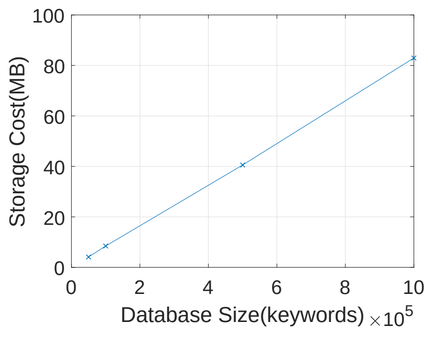


Fig. 9: Storage cost of MPT

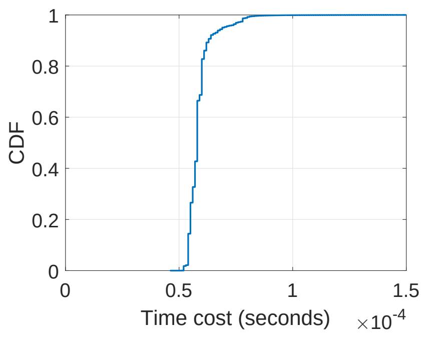


Fig. 10: *Generate* performance

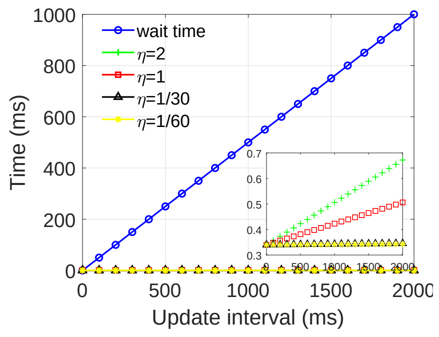


Fig. 11: *Verify* latency

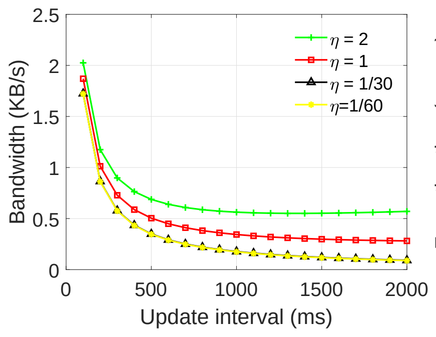


Fig. 12: Bandwidth consumption

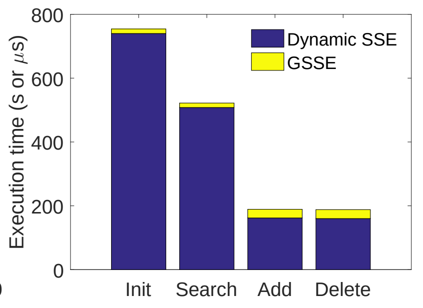


Fig. 13: Comparison with SSE [11]

AES-CBC extension and a 128 bytes RSA signature. Overall, the bandwidth of the authenticator includes two part: the overhead introduced by the fixed update time point and the overhead introduced by data update. We can observe that the bandwidth cost increases to about 2KB per second when the update interval decrease to zero, this is introduced by the fixed update time point which is inversely proportional to the bandwidth overhead. Moreover, the bandwidth gradually increases when the update interval becomes too long. This overhead is introduced by the length of the authenticator, because as the update interval grows, the length of the authenticator becomes larger. Overall, the cost should be acceptable to achieve GSSE. According to the results, in order to make a decent tradeoff between verification delays and bandwidth costs, we suggest choosing an update interval between 500 milliseconds and 1,500 milliseconds.

7.3 Comparison with Existing Schemes

We combine our verifiable SSE framework, i.e., the GSSE scheme, with a concrete implementation of dynamic SSE scheme proposed by Cash et.al [11], and show that the additional overhead introduced by GSSE scheme is not large.

In order to fairly compare the schemes, we test with the same dataset and parameters on our machine. As shown in Fig. 13, we measure the overhead of the initialization phase, the search phase and the update (add and delete) phase in both schemes. Here, the initialization phase is the *Init* operation of two million document-keyword pairs and the time unit is seconds. The measurement of the other three phases is the test of a single operation with a 10,000 keywords database and the time unit is microseconds. Note that the search phase in SSE corresponds to the *Prove* operation in GSSE. As seen from Fig. 13, our GSSE scheme introduces very little overhead. Compared to the initialization phase of the dynamic SSE scheme [11], the cost of the

Init operation in our GSSE scheme is significantly smaller, which just adds an extra 1.9% overhead. Moreover, for a single *Prove* operation, our GSSE scheme only introduces an additional overhead of 14 microseconds for the server, which adds an extra 2% compared to the search phase of [11]. For a single *Add* or *Delete* operation, our GSSE scheme only introduces 27 microseconds overhead, an extra 17% compared to corresponding add or delete phase in [11].

In Table. 3, we report the average communication overhead on the basis of 50,000 runs, due to the large variation of the search outcome. As a result, the average size of search results in SSE scheme [11] is about 53 kb, while our proof only needs 3 kb, which means the additional overhead introduced by GSSE is less than 6%. Moreover, the SSE scheme [11] generates 390 bytes search tokens on average, while our search tokens are only 32 bytes, which means the additional overhead is less than 9%. It shows that the overhead incurred by GSSE is acceptable.

TABLE 3: Comparison with the SSE scheme proposed by Cash et al. [11]

Communication cost	SSE [11]	GSSE
Search token	390 Bytes	32 Bytes
Search result/proof	53 Kilobytes	3 Kilobytes

8 Discussion

Multi-User Access Control. According to a well recognised survey [45], the architecture of searchable encryption includes four different types: single writer/single reader (S/S), single writer/multi-reader (S/M), multi-writer/single reader (M/S) and multi-writer/multi-reader (M/M). In this paper, we focus on the S/M architecture and provide search result verification for the multiple readers. We do not aim to develop a mechanism that achieves multi-user access control for encrypted search, since the access privilege of users can be well controlled by fine-grained access control mechanisms, such as role-based access control policies. The data owner can assign different roles for his/her users based on their responsibilities. Each role corresponds to an access privilege, such as read-only. Note that our scheme can readily be integrated into the SSE schemes that are enabled with enforced access control. There are many existing literatures [9], [33], [34] that studied how to enforce the user authorization via one-to-many encryption schemes, like broadcast encryption (BE) or attribute-based encryption (ABE) to control the sharing of these secret keys. For example, the Broadcast Encryption scheme [9] can securely transmit a message to all members of the authenticated users. A data owner sends a symmetric key K_U to a user U and meanwhile sends a state st_S to the server, where st_S is computed from the authenticated users group G and a symmetric key r. To search a keyword w, the data user should retrieve st_S from the server and recover r by using K_U. Then the data user sends the permutation $\Phi_r(\tau_w)$ to the server for access control. After receiving $\Phi_r(\tau_w)$, the server can recover the trapdoor τ_w by computing $\tau_w = \Phi_r^{-1}(\Phi_r(\tau_w))$. Note that, to revoke a user U, the data owner only needs to pick up a new key r' and calculated $st_S^{'}$ from the new group $G' = G \backslash U$. As the revoked user is no longer belong to the group G', the user can never recover the symmetric key r' from $st_S^{'}$, which means the user cannot generate a correct trapdoor. Therefore, by using broadcast encryption, a data owner does not need to re-encrypt the index after each revocation.

Generality. The GSSE scheme is a generic VSSE scheme since we separate the verification index from the index used for the searching operations in SSE. Therefore, the scheme can be applied to any SSE schemes and enable result verification for them. In addition, the GSSE scheme allows search result verification upon data update, i.e., enabling verifiability for the dynamic SSE schemes [10], [11], [17].

9 Conclusion

In this paper, we design GSSE, a dynamically verifiable SSE scheme, which can be applied to any SSE schemes with a three-party model and does not require modifications on them. By building authenticators and a proof index, GSSE provides efficient search result verification, while preventing data freshness attacks and data integrity attacks in SSE. The experimental results demonstrate that GSSE introduces acceptable overhead in verifying search results.

References

[1] A. Juels and B. S. Kaliski Jr, "Pors: Proofs of retrievability for large files," in *Proc. of CCS*, 2007, pp. 584–597.

[2] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proc. of Security and privacy in communication netowrks (SecureComm)*, 2008.

[3] S. Kamara, C. Papamanthou, and T. Roeder, "Cs2: A semantic cryptographic cloud storage system," Tech. Rep. MSR-TR-2011-58, Microsoft Technical Report (May 2011), http://research. microsoft. com/apps/pubs, Tech. Rep., 2011.

[4] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, "Enabling public auditability and data dynamics for storage security in cloud computing," *IEEE TPDS*, vol. 22, no. 5, pp. 847–859, 2011.

[5] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea, "Iris: A scalable cloud file system with efficient integrity checks," in *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2012.

[6] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Proc. of International Conference on Financial Cryptography and Data Security(FC)*, 2013.

[7] W. Sun, X. Liu, W. Lou, Y. T. Hou, and H. Li, "Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data," in *Proc. of INFOCOM*, 2015.

[8] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. of S&P*, 2000.

[9] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[10] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. of CCS*, 2012, pp. 965–976.

[11] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation." in *NDSS*, vol. 14, 2014, pp. 23–26.

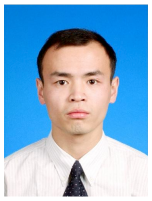
[12] Q. Wang, M. He, M. Du, S. S. Chow, R. W. Lai, and Q. Zou, "Searchable encryption over feature-rich data," *IEEE Transactions on Dependable and Secure Computing*, 2016.

[13] R. Bost, P.-A. Fouque, and D. Pointcheval, "Verifiable dynamic symmetric searchable encryption: Optimality and forward security." *IACR Cryptology ePrint Archive*, vol. 2016, p. 62, 2016.

[14] K. Kurosawa and Y. Ohtaki, "Uc-secure searchable symmetric encryption," in *Proc. of International Conference on Financial Cryptography and Data Security (FC)*, 2012.

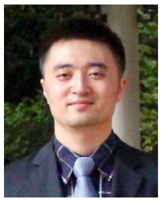
[15] Q. Chai and G. Gong, "Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers," in *Proc. of International Conference on Communications (ICC)*, 2012.
[16] K. Kurosawa and Y. Ohtaki, "How to update documents verifiably in searchable symmetric encryption," in *Proc. of International Conference on Cryptology And Network Security (CANS)*, 2013.
[17] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. of NDSS*, 2014.
[18] R. Cheng, J. Yan, C. Guan, F. Zhang, and K. Ren, "Verifiable searchable symmetric encryption from indistinguishability obfuscation," in *Proc. of AsiaCCS*, 2015.
[19] W. Ogata and K. Kurosawa, "Efficient no-dictionary verifiable sse," *IACR Cryptology ePrint Archive*, vol. 2016, p. 981, 2016.
[20] Y. Zhu, H. Hu, G.-J. Ahn, and M. Yu, "Cooperative provable data possession for integrity verification in multicloud storage," *IEEE TPDS*, vol. 23, no. 12, pp. 2231–2244, 2012.
[21] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. of CCS*, 2007.
[22] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," *ACM TISSEC*, vol. 17, no. 4, p. 15, 2015.
[23] "Merkle patricia tree," https://github.com/ethereum/wiki/wiki/Patricia-Tree, dec 13, 2016.
[24] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *Proc. of CRYPTO*, 1994.
[25] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," in *Proc. of the workshop on Cloud computing security (SCC)*, 2009.
[26] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proc. of EUROCRYPT*, 1987.
[27] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Authenticated hash tables," in *Proc. of CCS*, 2008.
[28] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.
[29] M. T. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an authenticated dictionary with skip lists and commutative hashing," in *Proc. of DARPA Information Survivability Conference & Exposition (DISCEX)*, 2001.
[30] Q. Zheng, S. Xu, and G. Ateniese, "Vabks: verifiable attribute-based keyword search over outsourced encrypted data," in *Proc. of INFOCOM*, 2014.
[31] P. Liu, J. Wang, H. Ma, and H. Nie, "Efficient verifiable public key encryption with keyword search based on kp-abe," in *Proc. of Broadband and Wireless Computing, Communication and Applications (BWCCA)*, 2014.
[32] Y. Yang, F. Bao, X. Ding, and R. H. Deng, "Multiuser private queries over encrypted databases," *International Journal of Applied Cryptography*, vol. 1, no. 4, pp. 309–319, 2009.
[33] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Outsourced symmetric private information retrieval," in *Proc. of CCS*, 2013.
[34] S.-F. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen, "An efficient non-interactive multi-client searchable encryption with support for boolean queries," in *Proc. of ESORICS*, 2016.
[35] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.
[36] "Rlp code," https://github.com/ethereum/wiki/wiki/RLP.
[37] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou, "Secure ranked keyword search over encrypted cloud data," in *Proc. of ICDCS*, 2010.
[38] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *Proc. of EUROCRYPT*, 2004.
[39] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Trans. on Communications*, vol. 39, no. 10, pp. 1482–1493, 1991.
[40] D. Mills, J. Martin, J. Burbank, and W. Kasch, "Network time protocol version 4: Protocol and algorithms specification," RFC 5905, June 2010.
[41] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Trans. on Computers*, vol. 100, no. 8, pp. 933–940, 1987.
[42] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *ACM Trans. on Operating Systems Review (SIGOPS)*, vol. 36, no. SI, pp. 147–163, 2002.
[43] D. Zhou and T. H. Lai, "An accurate and scalable clock synchronization protocol for ieee 802.11-based multihop ad hoc networks," *IEEE TPDS*, vol. 18, no. 12, pp. 1797–1808, 2007.
[44] "Enron_email dataset," https://www.cs.cmu.edu/~enron/, may 7, 2015.
[45] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, p. 18, 2015.

Jie Zhu is currently studying for Master degree of Computer Science in Graduate School at Shenzhen, Tsinghua University. She received her Bachelor degree in Beijing University of Posts and Telecommunications. Her current research interests include verifiable data search and privacy.

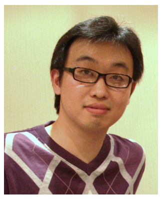


Qi Li received his Ph.D. degree from Tsinghua University. Now he is an associate professor of Graduate School at Shenzhen, Tsinghua University. He has ever worked at ETH Zurich, the University of Texas at San Antonio, The Chinese University of Hong Kong, Chinese Academy of Sciences, and Intel. His research interests are in network and system security, particularly in Internet and cloud security, mobile security, and big data security. He is currently an editorial board member of IEEE TDSC.

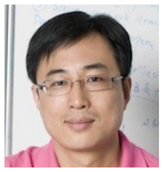
Cong Wang received the PhD degree in electrical and computer engineering from the Illinois Institute of Technology, in 2012. He is an assistant professor in the Computer Science Department, City University of Hong Kong. He worked at the Palo Alto Research Center in the summer of 2011. His research interests include the areas of cloud computing security, with current focus on secure data outsourcing and secure computation outsourcing in public cloud.



Xingliang Yuan received the B.S. degree from Nanjing University of Posts and Telecommunications in 2008, the M.S. degree from Illinois Institute of Technology in 2009, both in Electrical Engineering, and the Ph.D. degree in Computer Science from City University of Hong Kong in 2016. He is a Lecturer with the Faculty of Information Technology at Monash University. His research interests include cloud security, NFV security, and privacy-aware computing.



Qian Wang received the B.S. degree from Wuhan University, Wuhan, China, in 2003, the M.S. degree from Shanghai Institute of Microsystem and Information Technology (SIMIT), Chinese Academy of Sciences, Shanghai, China, in 2006, and the Ph.D. degree from Illinois Institute of Technology, Chicago, IL, USA, in 2012, all in electrical engineering.He is a Professor with the School of Computer Science, Wuhan University. His research interests include wireless network security and privacy, cloud computing security, and applied cryptography. He is currently an editorial board member of IEEE TIFS.



Kui Ren received the Ph.D. degree from the Worcester Polytechnic Institute. He is currently a Professor of Institute of Cyber Security Research in Zhejiang University. His current research interests span cloud and outsourcing security, wireless and wearable system security, and human-centered computing. He currently serves as an Associate Editor of IEEE TDSC, TMC, IEEE TSG, IEEE Wireless Communications, IEEE IoT Journal, Pervasive and Mobile Computing (Elsevier), and The Computer Journal (Oxford).