Publicly Verifiable Boolean Query Over Outsourced Encrypted Data
Shunrong Jiang, Xiaoyan Zhu, Linke Guo and Jianqing Liu

Abstract—Outsourcing storage and computation to the cloud has become a common practice for businesses and individuals. As the cloud is semi-trusted or susceptible to attacks, many researches suggest that the outsourced data should be encrypted and then retrieved by using searchable symmetric encryption (SSE) schemes. Since the cloud is not fully trusted, we doubt whether it would always process queries correctly or not. Therefore, there is a need for users to verify their query results. Motivated by this, in this paper, we propose a publicly verifiable dynamic searchable symmetric encryption scheme based on the accumulation tree. We first construct an accumulation tree based on encrypted data and then outsource both of them to the cloud. Next, during the search operation, the cloud generates the corresponding proof according to the query result by mapping Boolean query operations to set operations, while keeping privacy-preserving and achieving the verification requirements: freshness, authenticity, and completeness. Finally, we extend our scheme by dividing the accumulation tree into different small accumulation trees to make our scheme scalable. The security analysis and performance evaluation show that the proposed scheme is secure and practical.

Index Terms—Cloud Computing, Outsourced Encrypted Data, Query Integrity Verification

I. INTRODUCTION

The great flexibility and economic savings of cloud computing motivate companies and individuals to outsource their data to cloud servers. By outsourcing a dataset to the cloud, the data owner or other valid data users can then issue the cloud informational queries that are answered according to the dataset. This model captures a variety of real-world applications such as outsourced SQL queries, streaming dataset, and outsourced file systems. However, the privacy and confidentiality concerns of data often make them reluctant to do that. Therefore, it is natural for a data owner to encrypt data before outsourcing them to the cloud. As a result, the cloud server cannot reveal the content of the outsourced data. However, since data has been encrypted before outsourced the cloud, it obstructs the traditional data utilization service based on plaintext keyword search. Thus, how to efficiently obtain encrypted data from the cloud server is very important for outsourcing storage applications.

Motivated by this challenge, many searchable symmetric encryption (SSE) schemes [2–8] are proposed to satisfy different search functions on encrypted data. By using these schemes, a data owner can outsource encrypted data to protect the confidentiality of data, and data users can query from the cloud. Specifically, a pharmaceutical company would like to outsource a set of sanitized records of its clinical trials to the cloud server after encryption. Then, its employees or external third parties such as the European Medicines Agency use these information for research or other operations by keywords search [9]. However, due to the nature of the delegation/outsourcing, the cloud server can fully control the outsourced data and decide the SSE query result for the company or other third parties, which causes issues of trust.

There are several reasons for which the data owner/users cannot trust the cloud server: Firstly, the cloud server may run buggy software or its systems may be vulnerable to security breaches, leading to incorrect result [10, 11]. Secondly, in some applications, the company or third parties want to rule out accidental errors during the computation. Finally, when there is a legal dispute between the pharmaceutical company and a patient, the pharmaceutical company may collude with the cloud to provide an incomplete search result to the data user (such as a judge) [9]. Therefore, it is necessary for the cloud server to have the ability to provide proofs with which the data owner/users are able to verify the integrity of the search result returned by the cloud server.

As mentioned above, the query integrity is significantly important to the data owner/users. To achieve this purpose, we allow the data owner to perform some polynomial-time preprocessing on encrypted data before outsourcing them to the cloud and to save a small verification state that allows data users to verify the returned proof provided by the cloud server. When issuing an update operation, the data owner will update its verification state at the same time. If the verification state can be made by any third-party (not necessarily the user originating the search query), we say that the proof is publicly verifiable [9, 12]. Public verifiability is particularly important in multi-user settings.

The query integrity verification has been studied for structured attributed-value type database [13, 14] and streaming setting [15–18]. Many verification schemes are proposed to meet verification requirements (details are introduced in related work). However, these schemes may not be suitable for...
SSE condition: (1) These schemes assumed that the data stored at the third party publisher is in plaintext, while for SSE, the data is encrypted so that the publisher cannot “see” the actual content; (2) All of them are ordered by some sequences or the data update according to time slices which makes verification easier.

Obviously, the query integrity verification in SSE should contain three aspects [13]: (1) Freshness: The record values in the answer must be up-to-date; (2) Authenticity: Every record returned in the search result must originate from the data owner; (3) Completeness: Every record that satisfies the query condition must be in the search result. Moreover, the verification process should be secure and privacy-preserving. Even though there are many studies about the SSE, the query integrity verification work is limited except [19]. Moving a step forward, in this paper, we present a publicly verifiable integrity verification scheme which can publicly verify whether the cloud server has faithfully executed Boolean search operations in dynamic SSE (DSSE).

To realize query integrity verification in DSSE, a possible solution is that we can map query operations to set operations, so that the query integrity verification can be achieved by using the accumulation tree [20]. However, there are several challenges that must be overcome before we use this method: Firstly, how to map query operations to set operations in DSSE. Secondly, how to construct the accumulation tree with limited information of DSSE so that the construction process would not affect the architecture of DSSE. Thirdly, how to achieve the secure and privacy-preserving query integrity verification by using the accumulation tree in DSSE while ensuring the verification requirements. Finally, how to ensure the efficiency and scalability of the query integrity verification process in DSSE. All these challenges make the query integrity verification of DSSE different from the existing work. Thus, the contributions of our work are listed as follows:

- We propose a publicly Boolean query integrity verification scheme over the outsourced dynamic encrypted data to check the freshness, authenticity, and completeness of the query result.
- We construct an accumulation tree, a special Merkle hash tree, in DSSE to map Boolean operations of keywords to set operations, which is different from the traditional signature or aggregate signature.
- We extend our scheme by dividing encrypted data into small groups, then constructing the corresponding accumulation trees for each group to make our scheme efficient and scalable solutions.
- The security and performance analysis are carried out to show that the proposed scheme is privacy-preserving and practical.

The rest of this paper is organized as follows. Section II presents the system model and preliminaries. We describe the construction and the correctness of our scheme in Section III and Section IV, respectively. We give the security analysis in Section V and provide performance analysis in Section VI. The related work is given in Section VII. Finally, Section VIII concludes the paper.

### II. System model and Preliminaries

#### A. System Model and Motivation

A database $\mathbf{DB}=([\mathbf{d}_i], \mathbf{W})$ is a list of identifier and keyword-set pairs, where $\mathbf{d}_i$ is a document identifier and $\mathbf{W}_i$ is a list of keywords in that document. The system model is illustrated as Fig. 1 which contains three entities: a data owner $DO$, who outsources a large-scale collection of $d$ documents to the remote cloud server $S$; the cloud server $S$, which provides storage services; and data users $DUs$, who can enjoy the documents from the cloud. Considering the privacy problem and the efficient information retrieval, the cloud-based data storage and sharing process are described as follows:

- The data owner $DO$ first extracts the keywords of each document and builds a keyword index. Then $DO$ encrypts the documents as well as the keyword index. Notice that the documents are dynamic. That is, at any time the data owner can add, modify or delete one or more documents from the cloud.
- After the data owner outsources the encrypted documents as well as the encrypted keyword index to the cloud server, the data users $DUs$ can use the Boolean query expression to query and retrieve documents of interest from the cloud. To this end, the data users should first pass the authentication of the data owner, and then get an encrypted search token according to the Boolean query expression and the public verification key (VK).
- After receiving the query token, the cloud server $S$ executes the query and returns the encrypted documents according to the token. Moreover, to verify whether the cloud has correctly executed the search operation or not, an additional proof is also appended to the result.
- When receiving the result and the corresponding proof, the data users $DUs$ or others can verify the correctness of the search result by $VK$. Finally, $DUs$ can decrypt encrypted documents after the verification is correct.

---

1 We use “documents” generically; they can be text document, records in a relational database-in which case keyword are represented as attribute-value pairs, a combination of both, etc.
B. Attack Model

The data owners are naturally trusted. Both authorized and unauthorized data users are semi-trusted, meaning that they may try to infer some sensitive information from the query result and the corresponding proof. The cloud server is not trusted as it executes the search operations, which already implies that the cloud may manipulate the outsourced encrypted data. Moreover, we also consider potential malicious data users [15] which could collude with the cloud server or other malicious users, or help the cloud to cheat with other users. Note that the assumption about malicious data users enables the public verifiability property of our solution. Obviously, our attack model is more general.

C. Design Objective

The objective of our work is to design a publicly Boolean query integrity verification scheme in DSSE. First, we present a leakage function \( \mathcal{L} \), which covers all the information leakage in our scheme. For instance, the privacy leakage introduced by a Boolean query \( Q \) on DB is denoted as \( \mathcal{L}(DB, Q) \). Informally, the privacy leakage function for the query integrity verification in DSSE includes the following aspects:

- **Size Pattern:** The cloud server can learn the total number of data records in the database and the total number of search queries submitted by each data user [21];
- **Access Pattern:** The cloud server reveals the identifier of each encrypted data record that is returned for each query [21];
- **Search Pattern:** The cloud server can learn if the same encrypted data record is retrieved by two different queries [21];
- **Path Pattern:** The cloud server learns the set of children nodes \( v(d_{17}) \) of the accumulation tree in each query integrity verification.

Notice that the size pattern, access pattern, and search pattern are general information leakage in searchable encryption [3]. Path pattern is introduced for generating the proof for query integrity verification. We argue that revealing path pattern is only a minor leakage in the privacy-preserving Boolean query since the information we use to generate the proof is in the cloud server, which has little influence on the data privacy.

To sum up, the design objective is to achieve secure and privacy-preserving Boolean query integrity verification in DSSE under the leakage function \( \mathcal{L} \).

D. Preliminaries

1) **Bilinear Pairing:** Let \( G_1 \) be a multiplicative cyclic group of prime order \( p \), generated by element \( g \) \( \in G_1 \). Let \( G_T \) be a multiplicative cyclic group of the same order \( p \), such that there exists a pairing \( e : G_1 \times G_1 \rightarrow G_T \) with the following properties [22]:
   - **Bilinearity:** \( e(P^a, Q^b) = e(P, Q)^{ab} \) for all \( P, Q \in G_1 \) and \( a, b \in \mathbb{Z}_n \);
   - **Nondegeneracy:** \( e(g, g) \neq 1 \);
   - **Computability:** For all \( P, Q \in G_1 \), \( e(P, Q) \) is efficiently computable.

2) **Accumulation Trees:** An accumulation tree [20, 23, 24] \( AT \) is a tree with \([1/\epsilon]\) levels and \( m \) leaves, where \( \epsilon (0 < \epsilon < 1) \) is a parameter chosen upon setup. Each internal node of \( AT \) with degree \( m^\epsilon \) and level \( i \) in the tree contains \( m^{1-\epsilon} \) nodes (the root node of the tree lies in the maximum level). \( AT \) has a constant height for a fixed \( \epsilon \). Intuitively, it can be seen as a “flat” version of Merkle trees as shown in Fig. 2. Each leaf node contains the digest \( d(v_j) \) of the accumulation value \( acc(S_i) = g^{\sum_i e_i + s} \) for the set \( S_i \) where \( e_i \) is the element of \( S_i \) and \( s \) is the secret key. Moreover, each internal node contains the digest of the values of its children [20].

For the Boolean query, we can map the keyword Boolean operation to the set operation as shown in Fig. 3. At the same time, the set operation verification can be expressed by using the accumulation tree [20].

3) **Bilinear \( q \)-strong Diffie-Hellman assumption:** Let \( \lambda \) be the security parameter and let \( (p, G_1, G_T, e, g) \) be a uniformly random generated tuple of bilinear pairing parameters. Given the elements \( g, g^a, \cdots, g^{as} \in G_1 \) for some \( s \) chosen at random from \( \mathbb{Z}_q^* \), where \( q = poly(\lambda) \), there is no polynomial-time algorithm that can output the pair \( (a, e(g, g)^{1/(s+a)}) \in \mathbb{Z}_p^* \times \mathbb{Z}_p^* \) with probability \( \epsilon \).
III. PUBLICLY VERIFIABLE BOOLEAN QUERY OVER ENCRYPTED DATA

In this section, we define and construct our publicly verifiable scheme in DSSE. In order to introduce our scheme, we first describe the construction process of the query integrity verification with the SSE scheme, i.e., OSPIR-OXT [2].

A. Definition

Based on OSPIR-OXT, a publicly verifiable DSSE contains an algorithm \( \text{EDBSetup}(DB, RDK) \), an algorithm \( \text{Update}(EDB, AT) \), a protocol \( \text{GenToken}(K, \pi) \), a protocol \( \text{Search}(\text{token}) \), and an algorithm \( \text{Verify}(a(q), \Pi) \). The syntax is described as follows:

- \((K, s, EDB, AT) \leftarrow \text{EDBSetup}(DB, RDK)\). This algorithm can be divided into two phases: First, it takes a security parameter \( \lambda \) as input and outputs a secret key \( K \) and a secret key \( s \) for the accumulation tree construction \(^2\). Second, it takes a database \( DB \) as input, and outputs an encrypted database \( EDB \) \(^3\) and a constructed accumulation tree \( AT \) \(^4\). Then both \( EDB \) and \( AT \) are stored at the cloud server.

- \((EDB, AT) \leftarrow \text{Update}(EDB, AT)\). This algorithm runs between the data owner and the cloud server. The data owner inputs an update operation, the update index \( \text{ind}_i \) of the \( i \)th document, and the list of unique keywords in the document. The protocol adds or deletes the document from the EDB and updates the corresponding AT.

- \((\text{token}, VK_{\pi}) \leftarrow \text{GenToken}(K, \pi)\). This protocol is executed between the data owner and the user. For an authorized user with the Boolean query \( \pi = w_1 \cdots w_m \), the data owner uses \( K \) to generate the corresponding token to enable the search at the cloud server for the data user and the corresponding public verification key \( VK_{\pi} \).

- \((\Pi, a(q_i)) \leftarrow \text{Search}(\text{token})\). This protocol is run by the cloud server to conduct the search operation over encrypted index according to the token of the data user. The server returns the search result \( a(q) \) and the proof \( \Pi \). Notice that the proof can be an optional item [25].

- \((0, 1) \leftarrow \text{Verify}(a(q), \Pi, VK_{\pi})\). This algorithm is run by the verifier to verify whether the server has faithfully executed the search operations or not according to the search token by the proof and \( VK_{\pi} \).

Different from OSPIR-OXT, two new algorithms are added into our scheme. One is the \( \text{Update}(EDB, AT) \) algorithm which executes between the data owner and the cloud server; the other is the \( \text{Verify}(a(q), \Pi, VK_{\pi}) \) algorithm which is executed by data users. For the \( \text{Update}(EDB, AT) \) algorithm, the security and privacy concerns focus on the leakage of keywords in a document being added or deleted, which has little performance influence on private information retrieval (PIR) during the search phase. For the \( \text{Verify}(a(q), \Pi, VK_{\pi}) \) algorithm, the proof \( \Pi \) for query integrity verification is pre-calculated by the cloud server and then sent to the data user during the search phase. Hence, the \( \text{Verify}(a(q), \Pi, VK_{\pi}) \) algorithm should also leak little private information for PIR.

We also present the formal definition of the public verifiability [26]:

**Definition III.1.** (Publicly verifiable query security) Let \( VQ = (\text{EDBSetup}, \text{Update}, \text{GenToken}, \text{Search}, \text{Verify}) \) be a publicly verifiable query scheme.

Experiment \( \mathcal{E}_{\text{ExpPubVer}}^{\text{PubVer}}(VQ, DB, \lambda) \)

\[
(K, s, EDB, AT) \leftarrow \text{EDBSetup}(DB, \lambda)
\]

For \( i = 1, \cdots, l \), where \( l = \text{poly}(\lambda) \)

\[
\text{Verify}(a(q_i), \Pi_i, VK_{\pi_i}) \leftarrow \mathcal{A}(EDB, AT, V K_{\pi_i}, \Pi_i, a(q_i), V Q, \lambda)
\]

If \( a(q_i) \neq a(q_{i+1}) \) output “1” else “0”.

For any \( \lambda \in \mathbb{Z}_q^* \) and \( DB \), we define the advantage of an adversary \( \mathcal{A} \) making at most \( l \) queries in the above experiment against \( VQ \) as

\[
\text{Adv}_{\mathcal{A}}^\text{PubVer}(VQ, DB, l, \lambda) = Pr[\mathcal{E}_{\text{ExpPubVer}}^{\text{PubVer}}(VQ, DB, \lambda) = 1]
\]

A publicly verifiable query scheme \( VQ \) is secure for \( DB \) if

\[
\text{Adv}_{\mathcal{A}}^\text{PubVer}(VQ, DB, l, \lambda) \leq \text{neg}(\lambda)
\]

where \( \text{neg}(\cdot) \) is a negligible function of its input.

Before describing the details of our scheme, an overview and the OSPIR-OXT scheme will be presented.

B. Overview

The basic idea underlying the construction is that: to achieve the public Boolean query integrity verification for outsourced encryption documents, an accumulation tree is associated with encrypted documents as illustrated in Fig. 2. Then the data owner outsources both of them to the cloud server. During the search process, after executing the privacy-preserving Boolean query over encrypted data, the cloud should compute its corresponding proof according to the query result by using the accumulation tree. To compute the proof, the cloud maps relations of Boolean queries to the similar set operations of the accumulation tree. Finally, with the search result, the proof and the public verification key, the data user or others can verify the freshness, authenticity, and completeness of the search result even without decrypting them.
C. OSPIR-OXT

OSPIR-OXT [2] allows DU’s to search the outsourced database such that DU’s can only learn the information that DO authorizes them to learn while the cloud server still does not learn about the data or queried values as in the basic SSE setting. Furthermore, DO should learn as little as possible about queries performed by DU’s while still being able to verify the compliance of these queries according to the policy. We give a detailed description of OSPIR-OXT in Fig. 4.

Here, we set \( W = \bigcup_{i=1}^{d} W_i \), where \( d \) is the number of documents and \( W_i \) is a list of keywords in each document. DB(\( w \)) = \{\text{ind} i \text{ s.t. } w \in W_i \} and for each document in DB, a key rdk is used to encrypt that document where RDK is a list of document decryption keys. When a data user retrieves the index \( \text{ind} \) of a document matching his/her query, he/she also retrieves the record decrypting key rdk needed to decrypt that record. For each keyword \( w \in W \), an inverted index (corresponding to \( I(w) \)) referred as TSet(w) is built such that it points to all the ind values of documents in DB(w).

Each TSet(w) is identified by a string called stag(w). ind values in TSet(w) are encrypted under a secret key \( K_e \). The TSetSetup operation receives a collection \( T \) of lists \( T[w] \) for each \( w \in W \) and builds the TSet data structure out of these lists, then it returns TSet and a key \( K_e \). Next, TSetRetrieve operation can retrieve \( T[w] \) from TSet. According to the GenToken(\( K, \pi \)) algorithm, DU will get (\( \text{strap} ', \text{bstag}', \text{bxtrap}^2', ..., \text{bxtrap}^n' \)) and \( \text{env} = \text{AuthEnc}_{\text{K}_M}(\rho_1, ..., \rho_n) \) from DO to calculate the search token.

D. The Proposed Scheme

We take the conjunction keyword operation as an example to introduce our scheme and the details are described as follows:
System parameter. DO selects a bilinear group $e : G_1 \times G_2 \rightarrow G_T$, where $G_1$ and $G_2$ are two cyclic groups of prime order $p$ generated by an element $g$. Let $h(\cdot)$ be a hash function with range in $\mathbb{Z}_p \setminus \{1\}$ and $h(\cdot) = h(\cdot) \cdot Z_p^\ast$. Then DO chooses a random number $s \in Z_p^\ast$ as a secret key. Finally, DO publishes the system parameter $\text{param} = \{h(\cdot), p, G_1, G_2, e, g, g^s, q \geq \max\{m, \max_{x=l,\cdots,m}[\text{stag}(w_i)]\}\}$. Here $|\text{stag}(w_i)|$ means the number of documents in $\text{stag}(w)$ and $m$ is the number of keywords.

EDBSetup(DB, RDK)

Before outsourcing documents to the cloud, the data owner chooses key $K_s$, two vectors of elements $K_T = (k_1, \ldots, k_{m})$ and $K_X = (e_1, \ldots, e_m)$ at random in $Z_p^\ast$, key $K_f$ for PRF $F_p$ and key $K_{fr}$ for a symmetric authenticated encryption AuthEnc. $F_p$ and $F_r$ are PRF's outputs strings in $Z_p^\ast$ and $\{0,1\}^\tau$, respectively. $\tau$ is a security parameter. For generating the accumulation tree, DO chooses a key $sk = s$, and executes as follows:

- Firstly, DO initializes $XSet$ to an empty set and $T$ to an empty array which is indexed by group elements in $G_1$.
- Then for each $w_i = (l, v, al) \in W$, DO builds the tuple list $t$ and adds elements to set $XSet$ as follows:
  - Initialize $t$ to an empty list.
  - Set $\text{strap} \leftarrow (H(w_i))^{s_1}$, $\text{stag} \leftarrow (H(w_i))^{s_1}$, $(K_s, K_T) \leftarrow (F_r(\text{strap}, 1), F_r(\text{strap}, 2))$.
  - Initialize $c \leftarrow 0$; then for all $\text{ind}_d$ in $DB(w_i)$ in order:
    * Set $\text{rdk}_d \leftarrow \text{RDK}(\text{ind}_d)$.
    * $\text{tag} \leftarrow \text{Enc}(K_s, (\text{ind}_d | \text{rdk}_d))$.
    * Then $\text{c} \leftarrow c + 1$, $\text{z} \leftarrow F_p(K, c)$, $y \leftarrow \text{zind}$, $\text{zind}^{-1}$. Append $(\text{e}_i, y)$ to $t$.
  - Then $\text{tag} \leftarrow H(w_i)^{s_1}$, and add $\text{tag}$ to $XSet$.
  - Set $\text{acc}(w_i) = y^{1/\epsilon}$ as accumulation values associated with $\text{tag}(w_i)$.
- Then $T[\text{stag}] \leftarrow t$.
- DO creates $TSet \leftarrow TSetSetup'(T)$.
- After that, DO constructs the accumulation tree $\text{AT}$ as follows:
  - Initialize: DO picks a constant $\epsilon$, where $0 < \epsilon < 1$, and constructs $\text{AT}$ according to $\text{stag}(w_i)$ that has $l = [1/\epsilon]$ levels and $m$ leaves, where $m$ is the number of $\text{stag}(w_i)$.
  - Build: for each node $v$ of the tree, the algorithm recursively computes the digest $d(v)$ of $v$:
    * If $v$ is a leaf corresponding to keyword $w$, data owner sets $d(v) = \text{acc}(w_i)^{1/\epsilon}$.
    * If node $v$ is not a leaf, compute $d(v) = \prod_{u \in \text{children}(v)} d(u)$, where $\text{children}(v)$ denotes the set consisted by children nodes of $v$.
  - DO sets $d_0 = d(r)$ where $r$ is the root of $\text{AT}$, and keeps it to generate the public verification key $V_{\text{KP}}$.
- Finally, DO outputs key $K = (K_s, K_T, K_f, K_{fr})$ and EDB = $(TSet, XSet, AT, K_M)$.

Fig. 5: The Initialization of our scheme.

Update(EDB, AT)

To update document $\text{ind}_d$ to $\text{ind}_d'$, the update process is executed by DO as follows:

- DO computes $\text{rdk}_d \leftarrow \text{RDK}(\text{ind}_d)$, $\text{e}_d \leftarrow \text{Enc}(K_s, (\text{ind}_d | \text{rdk}_d))$, $\text{zind} \leftarrow F_p(K, \text{ind}_d)$, and $\text{tag} \leftarrow H(w_i)^{s_1}$, and retrievals $(\text{e}_i, y)$ according to $\text{e}_d$.
- Then DO computes $\text{rdk}_d' \leftarrow \text{RDK}(\text{ind}_d')$, $\text{e}_d' \leftarrow \text{Enc}(K_s, (\text{ind}_d' | \text{rdk}_d'))$, $\text{zind}' \leftarrow F_p(K, \text{ind}_d')$, and $y' = \text{zind} \cdot \text{zind}'^{-1}$.
- Then $\text{tag} \leftarrow H(w_i)^{s_1}$, and add $\text{tag}$ to $XSet$.
- Then DO updates the corresponding accumulation tree $\text{AT}$ as follows:
  - After computing $\text{e}_d$, set $w_0$ to be the leaf node of $\text{AT}$ corresponding to $\text{stag}(w_i)$.
  - Let $v_0, v_1, \ldots, v_l$ be the path in $\text{AT}$ from node $v_0$ to the root of the tree, where $l = [1/\epsilon]$.
  - Then, DO sets $d'(v_0) = \text{acc}(w_i)^{1/\epsilon}$; for document $\text{ind}_d$ is deleted from $w_i$, data owner $d'(v_0) = \text{acc}(w_i)^{1/\epsilon}$; for document $\text{ind}_d$ is added to $w_i$, data owner $d'(v_0) = \text{acc}(w_i)^{1/\epsilon}$; for document $\text{ind}_d$ is added to $w_i$, and keeps it to generate the public verification key $V_{\text{KP}}$.
- At last, DO replaces $d(v_j)$ by $d'(v_j)$ from $v_0$ to the root of $\text{AT}$.

Fig. 6: The update process of our scheme.

1) Initialization: The basic purpose of this phase is to initialize the system of our scheme which mainly contains two parts: the SSE construction and the corresponding accumulation tree construction. For the SSE construction, the data owner encrypts outsourced documents and generates the document index for SSE. To verify the Boolean queries results, an accumulation tree is built according to the document index. The construction is shown in Fig. 5. To support update, we should order the index in each keyword. Notice that the process is privacy-preserving, since we only use $e_d$ computed by SSE to construct our accumulation tree.

Notice that, to generate the proof according to different keywords, $XSet$ is divided into different sets according to $\text{stag}(w_i)$ during the accumulation tree construction. Moreover, in the Search(token) algorithm, token is in the form of $\text{stag} \leftarrow H(w_i)^{s_1}$, so we also need to classify $\text{tag}$ according to $\text{stag}$. Actually, this information can be learned by the cloud server through the search operation.

2) Update: In a DSSE scheme, we need to support functions of adding, modifying, and deleting documents. To update $\text{ind}_d$ with $\text{ind}_d'$, the data owner computes $TSet$, $XSet$, $AT$ using $\text{ind}_d$, then retrieves them from the cloud and replaces them by $TSet'$, $XSet'$, $AT'$ computed with $\text{ind}_d'$. However, a subsequent problem is the privacy leakage during the update process. To protect the privacy, the update of index and accumulation tree should leak as little information about the access pattern as possible. The access pattern during the update is that server would be allowed to learn whether two newly added documents share a keyword or not, as soon as they are added and before such a keyword is searched for (instead of...
learning whether they share keywords with the original set of documents that were added when initializing the system.

To reveal very little information beyond this, a lazy deletion strategy [5] is proposed. In lazy deletion strategy, the index document of a keyword (for the original set of documents) is not updated until that keyword is searched for to minimize the privacy leakage during the update process. However, in the OSPIR-OXT, token is “blind” for the data owner, even the data owner does not know the search keywords of data users. Therefore, we can only update the TSet, XSet and AT immediately without considering protection of the access pattern. Actually, in SSE, the keyword access pattern and document access pattern are revealed. It means that if the same keyword is searched for multiple times or if the same document appears in multiple keyword searches, the server learns about that. Techniques for hiding this information take significant costs [6]. Finally, the update process is given in Fig. 6. DO updates the corresponding AT to AT′ as shown in Fig. 7. It is clear that in the case that document indi is deleted from wi, d′(v0)=acc(wi)[ωi+s+1]; in the case that indi is added to wi, d′(v0)=acc(wi)[ωi+s].

3) Token Generation: As described in OSPIR-OXT, the token generation of PIR should let the data owner learn about queries as little as possible while it is still able to verify the compliance of these queries to his/her policy. To achieve this, the data user blinds his query keywords and uses the attributes policy to ensure the valid identity to generate the proof. Then, AT computes the conjunction query proof Pj using the extended Euclidean algorithm where Pj is computed in $W_{i,j}$.

Algorithm 1: The proof generation of the query result.

Require: $\mathbb{w} = \{w_1, \ldots, w_n\}$, $\mathbb{I} = \{\mathbb{e}_1, \mathbb{e}_2, \ldots, \mathbb{e}_k\}$ and g.

1: for $i = 1, \ldots, n$ do
2: Let $v_0$ be the leaf node corresponding to stag($w_i$) of the accumulation tree AT and $v_{l_1}, \ldots, v_{l_{\lceil \log_2 n \rceil}}$ be the node path from $v_0$ to the root $r$.
3: for $j = 1, \ldots, \lceil \log_2 n \rceil$ do
4: Compute $\gamma_j = g^{\prod_{s \in \text{stag}(w_i)}(-)(h(d(\tau_{s}))+s)}$.
5: end for
6: Cloud outputs $\pi_i = (d(v_0), \gamma_1, \ldots, (d(v_{l_1})-1, \gamma_{l_{\lceil \log_2 n \rceil}})$ as the freshness proof of acc(wi).
7: Cloud computes $W_{i,j} = g^{\prod_{s \in \text{stag}(w_i)}(-)(s+2)}$ as subset proof of stag($w_i$).
8: Cloud computes $F_{i,j} = g^{q_{j}(s)}$ according to Lemma III.2 as operation completeness proof.
9: Cloud outputs the proof $\Pi = (\pi_i, W_{i,j}, F_{i,j})$.
10: end for

- The query result is the subset of stag($w_i$). We compute the subset witness $W_{i,j}$ as subset proof. Since $s$ is a secret number which is only known by the data owner, we can use the Lemma III.1 to compute $P_j(s)\leftarrow \prod_{s \in \text{stag}(w_i)}(x+s)$. Then, we compute $W_{i,j}$ by using $g$. The computation complexity is $O(d \log \tilde{d})$ where $\tilde{d}$ is the number of documents.
- The query result is complete. The completeness witness $F_{i,j}$ is calculated as Lemma III.2 by using the extended Euclidean algorithm where $P_j(s)$ is computed in $W_{i,j}$.

Lemma III.1. (Polynomial interpolation with FFT [27]) Let $\prod_{j=1}^{n}(s+x_j) = \sum_{k=0}^{n} a_k s^k$ be a degree-$n$ polynomial. Given $x_1, x_2, \ldots, x_n$, the coefficients $a_n, a_{n-1}, \ldots, a_0$ can be computed with $O(n \log n)$ complexity.

Lemma III.2. Assume set $l$ is the intersection of sets $S_1, S_2, \ldots, S_i$, if and only if there exists polynomials $q_1(s)P_1(s) + q_2(s)P_2(s) + \cdots + q_i(s)P_i(s) = 1$. Moreover, the complexity of computing polynomial $q_1(s), q_2(s), \ldots, q_i(s)$ is $O(d \log^2 \tilde{d} \log \tilde{d})$.
Each element $e_i$ belongs to some sets $\text{stag}(w_i)$ by computing $W_{\text{stag}(w_i)} = g^{\prod_{e_j \in \text{stag}(w_i)} (\delta_i + \varepsilon_j)}$. The computation complexity is $O(h)$.

Each $\text{stag}(w_i)$ is a subset of the query result $U$. The subset witness $W_{U,i}$, for $i = 1, \ldots, n$, is computed as $W_{U,i} = g^{\prod_{e_j \in U-\text{stag}(w_i)} (\delta_i + \varepsilon_j)}$ by using Lemma III.1. The computation complexity is $O(d \log \hat{d})$.

As for the subtraction operation query result, the subset can be expressed as $S = S_j, S_i = S_j \cap S_i$, where $S_j \cap S_i$, means the conjunction set of $S_j$ and $S_i$. To verify the correctness of the subtraction operation, we only need to verify the correctness of $S_j \cap S_i$, which has been discussed before.

5) Verification: On reception the search result $a(q)$ and the corresponding proof $\Pi$, the query integrity verification is executed by any verifier with $a(q), \Pi$, and $V K_{\text{acc}}$ as follows:

As for the conjunction query result $I = \{e_1, e_2, \ldots, e_h\}$, $\Pi$ and $V K_{\text{acc}}$, the verification process consists of following steps:

Verify the freshness of accumulation values by checking following equations:

\[ e(d(v_0), g)^{e(\text{acc}(I), W_{I,j})} \] (3)

At last, we execute the completeness verification if the subset verification is correct:

\[ \prod_{j=1}^{n} e(W_{I,j}, F_{1,j}) = e(g, g). \] (4)

If the equation holds, we accept the returned query result $I$. For union operations, after computing the coefficients $\delta_1, \ldots, \delta_h$ and verifying the accumulation values’ proof $\pi_i$, we execute the verification process as follows:

Element verification is executed by checking:

\[ e(W_{\text{stag}(w_i), g}, g^{\varepsilon_j}) = e(\text{acc}(w_i), g). \] (5)

We verify whether all sets $\text{stag}(w_i)$ are subsets of the union $U$ or not by checking the following condition:

\[ e\left(\prod_{i=0}^{h} (g^s)^{\delta_i}, g\right) = e(\text{acc}(U), g)\]. (6)

If all the verifications are correct, we accept the returned query result.

Notice that the freshness verification can be achieved using Merkle tree which is much more efficient than using an accumulation tree. We will not discuss it in this paper, since the generation of the proof is the major overhead, especially when the dataset is scalable.

![Fig. 8: We divide the encrypted index into different groups. Group $j$ is ended by $e_{j\mu} (\zeta = \mu \Delta).$ $\text{Stag}(w_i)$ in group $j$ is defined as $\text{Stag}(w_{i,j})$ and the corresponding accumulation tree $AT_j$ is computed for group $i$.](image)

E. Extension

It is obvious that computing $F_{1,i}$ requires running the Extended Euclidean (XGCD) algorithm, whose time increases drastically with the degree of polynomials. To address this problem, a natural solution is to reduce the degree of polynomials. Moreover, the cloud server can run in parallel. Combining with these two conditions, we divide all encrypted $e_i$ into different groups and construct accumulation trees as shown in Fig. 8. We first order $e_i$ in each keyword $w_i$, then, for all $e_i$ in keyword $w_i$, we divide them into different groups, where the member $e_i$ in group $j$ is between $(j-1)\mu$ and $j \mu$. We use $\text{Stag}(w_{i,j})$ to denote the set containing all members in group $j$ of the keyword $w_i$ and compute the corresponding accumulation tree $\text{acc}(w_{i,j})$. At last, we compute the accumulation tree $AT_j$ for group $j$ by using $\text{acc}(w_{i,j})$ where $1 \leq i \leq m$.

In this way, the degree of polynomials used to compute each accumulation value is lower than the basic scheme. We assume that each keyword $w_i$ contains the same number $n_w = 10,000$ of ind$e_i$ and $e_i$ in each keyword follows uniform distribution $\theta$. We divide the encrypted index into 10 groups. The result of the conjunction keyword query operation \"$w_1$ AND $w_2$ AND $\cdots$ AND $w_m$\" is $\{e_1, e_2, \ldots, e_h\}$. The freshness verification process almost keeps the same. Only the correctness of the subset and the corresponding completeness verification are related to the degree of polynomials. The best condition is that the result is in the same group $j$. In this way, the verification process almost keeps the same as the basic scheme while each degree of polynomials decreases. The worst situation is that the result disperses in all groups. In this way, the verification process should be executed in each group which needs 10 times computation overhead comparing with the basic scheme.

6Actually, the extension scheme is more suitable for the structured database that each item contains all the keywords such as the database used in [28].
However, the degree of polynomials in each group becomes lower, which makes the dominant factor of the cost sharply cut off. In addition, the verification process can be executed in parallel. A detailed performance discussion of our extension scheme will be given in the performance analysis.

IV. CORRECTNESS

We show the correctness of the verification process as follows:

After receiving $\pi_i$, the freshness of the accumulation value verification is executed:

$$e(d(v_0), g) = e(\text{acc}(w_i)^{i+s}, g) = e(\text{acc}(w_i), g^s),$$ (7)

and for $j = 1, \cdots, \lceil 1/e \rceil$,

$$e(d(v_j), g) = e(g^{\Pi_{\exists N(v_j)}(h(d(\pi)+s))}, g) = e(g^{\Pi_{\exists N(v_j)} - (v_j-1)(h(d(\pi)+s))}, g^{h(d(v_j-1)+s)}) = e(\gamma_j, g^{h(d(v_j-1)+s)}) = e(\gamma_j, g^{h(d(v_j-1))}g^s).$$ (8)

The correctness of the subset verification is described as follows:

$$e(\prod_{i=0}^{h}(g^s)^{\delta_i}, W_{i,j}) = e(g^{\sum_{i=0}^{h} \delta_i s^i}, W_{i,j}) = e(g^{\Pi_{\exists stag(w_j) - (x+s)}}, W_{i,j}) = e(g^{\Pi_{\exists stag(w_j)}}(x+s), g) = e(\text{acc}(w_j), g).$$ (9)

Finally, the correctness of the completeness is given:

$$\prod_{j=1}^{n} e(W_{i,j}, F_{i,j}) = e(g, g)^{\sum_{j=1}^{n} q_i(s)P_i(s)} = e(g, g).$$ (10)

V. SECURITY ANALYSIS

In this section we give our game-based definition of security, and then prove that our protocols satisfy this definition.

Setup: The challenger runs $\text{GenKey}(1^A)$, and generates public parameters $\text{param}$ which are given to the adversary. The challenger keeps the private keys $K$ and $s$ to herself/himself.

Phase 1: The adversary adaptively issues a number of requests. Each request belongs to one of the following types:

- Encryption request (DB): For the ciphertext request, the adversary outputs a database $DB$. The challenger responds with an encrypted database $EDB$.
- Accumulation tree request (EDB): For the accumulation tree request, the adversary outputs an encrypted database $EDB$. The challenger responds with an accumulation tree $AT$ with $s$.
- Token request ($\pi$): For the token request, the adversary outputs a Boolean query $\pi$. The challenger responses with the query token $\pi$ and the public verification key $VK_{\pi}$.
- Boolean query request (token): For the Boolean query request, the adversary outputs a token. The challenger responds with the query result $a(q)$ and the corresponding proof $\Pi=(\pi_i, W_{i,j}, F_{i,j})$.

Once the adversary decides that Phase 1 is over, she/he chooses a challenge as Proof challenge:

Proof challenge: The adversary outputs an encrypted database $EDB$, a query token $\pi$, a public verification key $VK_{\pi}$, and a query result $a(q)$. The challenger re-executes the query operation according to $EDB$, $\pi$, and $VK_{\pi}$ to verify the validity of $a(q)$. If $a(q)$ is valid, the challenger computes the corresponding proof $\Pi=(\pi_i, W_{i,j}, F_{i,j})$.

Phase 2: The adversary issues more queries. The challenger responds as in Phase 1.

Guess: The adversary outputs one or more guesses among three guesses for attacking freshness, authenticity, and completeness, respectively.

- Freshness guess: The adversary outputs $(EDB, \pi, \{a(q)^*\}, VK_{\pi})$, such that $a(q)^*$ is not the fresh result of query token over $EDB$, while $\pi^{*}$ is an acceptable proof for freshness of result $a(q)^*$. Notice that $\pi^{*}$ was not the same as given in Proof challenge. As is defined, $\pi^{*}$ was neither generated in Phase 1 nor in Phase 2. In other words, the adversary can demonstrate that a non-fresh answer can pass the freshness verification.
- Authenticity guess: The adversary outputs $(EDB, \pi, a(q)^*, VK_{\pi})$, such that $a(q)^*$ is not the correct result of query token over $EDB$, while $\pi^{*}$ is an acceptable proof for authenticity of result $a(q)^*$. Notice that $W_{i,j}$ is not the same as given in Proof challenge. As is defined, $W_{i,j}$ was neither generated in Phase 1 nor in Phase 2. In other words, the adversary can demonstrate that a wrong answer can pass the authenticity verification.
- Completeness guess: The adversary outputs $(EDB, \pi, a(q)^*, VK_{\pi})$, such that $a(q)^*$ is not the complete result of query token over $EDB$, while $\pi^{*}$ is an acceptable proof for the completeness of result $a(q)^*$. Notice that $F_{i,j}^{*}$ was not the same as given in Proof challenge. As is defined, $F_{i,j}^{*}$ was neither generated in Phase 1 nor in Phase 2. In other words, the adversary can demonstrate that an incomplete answer can pass the completeness verification.

Theorem V.1. Our public verification protocol for the Boolean query over outsourced encrypted data preserves the freshness, authenticity, and completeness properties under the bilinear $q$-strong Diffie-Hellman assumption.

Proof: The main idea of the following proof is to simulate the query security game under our proposed scheme as defined above. Let $A$ be the adversary that has advantage against our freshness, authenticity, or completeness of the verification protocol. Let us construct an adversary $B$ that uses $A$ to gain advantage against $q$-strong Diffie-Hellman assumption. The adversary $B$ acts as the challenger for $A$ and uses $A$’s outputs as her/his own outputs. $B$ answers $A$’s queries as follows:

Setup: The $B$’s challenger runs $\text{GenKey}(1^B)$, and generates public parameters $\text{param} = \{b(h), p, \mathbb{G}_1, \mathbb{G}_T, e, g, g^2\}$ which are then given to $B$. The challenger keeps the private keys $K$ and $s$ to itself. Then $B$ gives $\text{param}$ to $A$. 

2168-7161 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Phase 1: The adversary \( A \) adaptively issues a number of requests to \( B \), where each request belongs to one of the following types:

- Encryption request (DB): For the ciphertext request, \( A \) outputs a database \( DB \) to \( B \). Since \( B \) does not have \( K \), she/he submits an encryption request (DB) to her/his challenger. The challenger runs \( Enc(DB, RDK, K) \) (which is \( EDBSetup(DB, RDK) \) in OSPIR-OXT) and gives \( EDB \) to \( B \). Then \( B \) gives \( EDB \) to \( A \).
- Accumulation tree request (EDB): For the accumulation tree request, \( A \) outputs an encrypted database \( EDB \) to \( B \). Since \( B \) does not have \( K \), she/he submits an accumulation tree request (EDB) to her/his challenger. The challenger runs \( Acc(EDB, s) \) and gives \( AT \) to \( B \). Then \( B \) gives \( AT \) to \( A \).
- Token request (\( \pi \)): For the token request, \( A \) outputs a Boolean query \( \pi \) to \( B \). Since \( B \) does not have \( K \), she/he submits a token request (\( \pi \)) to her/his challenger. The challenger runs OSPIR-OXT.GenToken(\( K, \pi \)) algorithm and gives token to \( B \). Moreover, the challenger generates \( V K_{\pi}=\{h(p), g, G_1, G_T, e, g, g_0\} \) to \( B \). Finally, \( B \) gives token and \( V K_{\pi} \) to \( A \).
- Boolean query request (token): For the Boolean query request, \( B \) runs an encryption request, an accumulation tree request, and a token request on \( DB \) to obtain the encrypted \( EDB \), the corresponding accumulation tree \( AT \), token, and \( V K_{\pi} \) according to \( \pi \). Then \( B \) runs OSPIR-OXT:Search(token) algorithm to get the query result \( a(q) \) and computes the corresponding proof \( \Pi=(\pi_i, W_{i,1}, F_{i,1}) \). Finally, \( B \) gives \( a(q) \) and \( \Pi \) to \( A \).

Once \( A \) decides that Phase 1 is over, she/he chooses a challenge as Proof challenge:

**Proof challenge:** \( A \) outputs an encrypted database \( EDB \), an accumulation tree \( AT \), a query token \( \pi \), a public verification key \( V K_{\pi} \), and a query result \( a(q) \). \( B \) re-executes the query operation according to \( EDB \), \( AT \), \( \pi \), and \( V K_{\pi} \) to verify the validity of \( a(q) \). If \( a(q) \) is valid, \( B \) computes the corresponding proof \( \Pi=(\pi_i, W_{i,1}, F_{i,1}) \).

Phase 2: The adversary \( A \) issues more queries. \( B \) responds as in Phase 1.

**Guess:** The adversary \( A \) outputs one or more guesses among three guesses for attacking freshness, authentication, and completeness, respectively.

**Freshness guess:** \( A \) outputs \( EDB, AT, \pi, V K_{\pi}, a(q^*) \), such that \( a(q^*) \) is not the fresh result of query token over \( EDB \), while \( \Pi^*=(\pi_i^*) \) is an acceptable proof for freshness of result \( a(q^*) \). In other words, the values \( acc(w_1), acc(w_2), \cdots, acc(w_n) \) and \( \pi_i^*, \pi_j^*, \cdots, \pi_n^* \) (where \( \pi_i^*=(d(v_0)+\gamma_i^*), \cdots, (d(v_{[1/\epsilon]}-1)+\gamma_i^*[1/\epsilon]) \)) picked by \( A \) can pass the freshness verification. This implies that: for each \( w_j \)

\[
e(d(v_0)+\gamma_i^*, g) = e(acc(w_1), g) = e(acc(w_2), g^*\gamma_i^*)
\]

and for \( j = 1, \cdots, [1/\epsilon] \),

\[
e(d(v_j)+\gamma_i^*, g) = e(g^{(\Pi\pi_{N}(v_j)+h(d(v_j)+\gamma_i^*))}, g) = e(g^{(\Pi\pi_{N}(v_j)+h(d(v_j)+\gamma_i^*))}, g^{h(d(v_j)-1)+\gamma_i^*}) = e(\gamma_j^*, g^{h(d(v_j)-1)+\gamma_i^*}) = e(\gamma_j^*, g^{h(d(v_j)-1)+\gamma_i^*})
\]

Notice that, the root of \( \Pi^* \) \((d_0=\pi_{v_1}) \) is in \( V K_{\pi} \). Thus, we have

\[
e(d(v_{[1/\epsilon]}), g) = e(\gamma_{[1/\epsilon]}, g^{h(d(v_{[1/\epsilon]})-1)+\gamma_i^*})
\]

Since for each \( w_i \), we have \( acc(w_i) \neq acc(w_i) \) and \( acc(w_i) \) passes the freshness verification, which implies that \( B \) has found \( acc(w_i) \) such that

\[
e(acc(w_i), g^*\gamma_i^*) = e(g^{(\Pi\pi_{N}(w_i)+\gamma_i^*)}, g).
\]

Thus, we have

\[
e(acc(w_i), g)^{(i+s)} = e(g^{\Pi\pi_{N}(w_i)+\gamma_i^*}, g)^{(i+s)}
\]

According to the proof of [23], if \( A \) can achieve these, it means \( B \) can break the bilinear \( q \)-strong Diffie-Hellman assumption for the setting \((p, G_1, G_T, e, g, g_0)\) in \( V K_{\pi} \). Thus, if \( A \) has advantage \( \varepsilon_1 \) in this attack, \( B \) has advantage \( \varepsilon_1 \) in breaking the bilinear \( q \)-strong Diffie-Hellman assumption.

**Authenticity guess:** \( A \) outputs \( EDB, AT, \pi, V K_{\pi}, a(q^*), \Pi^* \), such that \( a(q^*) \) is not the correct result of query token over \( EDB \), while \( \Pi^*=(W_{i,j}^*) \) is an acceptable proof for authenticity of result \( a(q^*) \). In other words, the values \( acc(w_1), acc(w_2), \cdots, acc(w_n) \) and \( W_{i,1}, W_{i,2}, \cdots, W_{i,n} \) picked by \( A \) can pass the authenticity verification as follows:

\[
e(\prod_{i=0}^h (g^s)^{\delta_i}, W_{i,j}^*) = e(acc(w_j), g).
\]

Since this guess implies \( acc(w_j) = acc(w_j) \), it means \( A \) can achieve

\[
e(g^{\Pi\pi_{N}(w_i)+\gamma_i^*}, W_{i,j}^*) = e(acc(w_j), g) \land l \not\subseteq w_j
\]

for all \( j = 1, \cdots, n \). If \( A \) has advantage \( \varepsilon_2 \) in this attack, \( B \) has advantage \( \varepsilon_2 \) in breaking the bilinear \( q \)-strong Diffie-Hellman assumption.

**Completeness guess:** The adversary outputs \( EDB, AT, \pi, V K_{\pi}, a(q^*), \Pi^* \), such that \( a(q^*) \) is not the complete result of the query token over \( EDB \), while \( \Pi^*=(F_{i,j}^*) \) is an acceptable proof for completeness of result \( a(q^*) \). In other words, the values \( W_{i,1}, W_{i,2}, \cdots, W_{i,n} \) and \( F_{i,1}, F_{i,2}, \cdots, F_{i,n} \) picked by \( A \) can pass the completeness verification. This implies \( e(g^{\Pi\pi_{N}(w_i)+\gamma_i^*}, W_{i,j}^*) = e(acc(w_j), g) \land l \not\subseteq w_j \) for \( j = 1, \cdots, n \). It is equivalent to writing \( W_{i,j}^* \) as the subset witness \( W_{i,j}^* \) and \( W_{i,j}^* = g^{\prod_{i=0}^{\pi_{N}(w_j)}(x+n)} \). This implies that \( P_1(s), P_2(s), \cdots, P_n(s) \) have at least one common factor, such as \((s+r)\) and \( P_j(s) = (s+r)Q_j(s) \) it holds for some polynomials \( Q_j(s) \) which are computable in polynomial time for \( j = 1, \cdots, n \). Thus,
Therefore we can derive the $1/(s + r)$ of $e(g, g)$ as $e(g, g)^{-\frac{1}{s+r}} = \prod_{j=1}^{n} e(g^{Q_j(s)}, F_{1,j}^*)$ in the following aspects: (1) First, the data user verifies all query results are in the same group. (2) Second, the verification proof consists of the freshness proof of the query keywords ($\epsilon$ is constant when the accumulation tree is built). Therefore, the total verification cost for proof is $\frac{1}{\epsilon} + 1) = (1/\epsilon + 1) - n||G_1||$ bits. While for the extended scheme, the total verification cost is $\frac{1}{\epsilon} + 1) = (1/\epsilon + 1) - n||G_1||$ bits where $\epsilon$ is the best situation that all query results are in the same group.

4) Proof Generation Cost: As described in Algorithm 1, the verification proof consists of the freshness proof $\pi_i$, the subset proof $W_i$, and the completeness proof $F_i$. To compute $\pi_i$, it consumes $n(1/\epsilon + 1) = (1/\epsilon + 1) - n||G_1||$ bits. While for the extended scheme, the total verification cost is $\frac{1}{\epsilon} + 1) = (1/\epsilon + 1) - n||G_1||$ bits. To compute $F_i$, the cloud should first use Lemma III.2 to compute $q_i(s)$, whose computation complexity is about $O(n \sum_{i=1}^{n} (|w_i| - h) log^2(|w_i| - h) log log(|w_i| - h))$. Then, for each computed $q_i(s)$, the cloud should compute $F_i \cdot g^{q_i(s)}$ which consumes $nT_{EXP}$ computation overhead. Thus, the total computation overhead for $F_i$ is $O(n \sum_{i=1}^{n} (|w_i| - h) log^2(|w_i| - h) log log(|w_i| - h))(T_{mul} + T_{add} + nT_{EXP})$.

For the extended scheme, to compute $W_i$, it needs $\frac{1}{\epsilon} + 1) = (1/\epsilon + 1) - n||G_1||$ bits. The total computation overhead for $F_i$ is $O(n \sum_{i=1}^{n} (|w_i| - h) log^2(|w_i| - h) log log(|w_i| - h))(T_{mul} + T_{add} + nT_{EXP})$.

5) Verification Cost: The verification cost consists of the following aspects: (1) First, the data user verifies the accumulation values which consume $2(1/\epsilon + 1)T_{PAIR}$; (2) Second, we calculate the coefficients of the returned result. The time cost for computation is $(T_{mul} + T_{add})h$; (3) Third, the algorithm checks the subset condition which takes a period time of $hT_{EXP} + 2nT_{PAIR}$; (4) Finally, it checks the completeness condition whose time consumption is $nT_{PAIR} + (n - 1)T_{MUL}$. To sum up, the total verification cost is $(2(1/\epsilon + 1) + 3)nT_{PAIR} + (T_{mul} + T_{add})h + (n - 1)T_{MUL} + hT_{EXP}$. While for the extended scheme, the total verification cost is $2n(1/\epsilon) + 3n + 3nT_{PAIR} + (T_{mul} + T_{add})h + (n - 1)T_{MUL} + hT_{EXP}$.
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2684811, IEEE Transactions on Cloud Computing

We observe that the overhead increases almost linearly with the number of keywords \( m \) when the number of keywords \( m \) is 9 and 64, respectively. It shows that, with the number of conjunction keywords increasing, the computation overhead is influenced by the number of index/documents \( n_w \) in each keyword and the number of conjunction keywords \( n \). Given parameter \( m=64, \epsilon=1/3, n_w=10,000 \) and \( h=0.1n_w \), we can see that to generate the proof, it consumes about 430s and 1,740s for \( n=4 \) and \( n=16 \), respectively. The larger the number of conjunction keywords \( n \) is, the more time this proof generation phase takes. The reason leading to such result is that with the number of conjunction keywords increasing, the number of the set check process required to be executed grows correspondingly. It needs more exponentiation operations and polynomial expansion operations while the number of index \( n_w \) and \( h \) in each keyword are the same.

Similar to the relation between the number of conjunction keywords \( n \) and the proof generation time, as the number of index/documents \( n_w \) in each keyword increases, the proof generation time becomes longer, and the extent of improvement gradually levels up. For scenario that the number of conjunction keywords \( n=16 \), the proof generation time increases from 0.4s to 1,740s when \( n_w \) increases from 100 to 10,000. This is because computing \( F_{i,j} \) requires running the XGCD algorithm, whose time increases drastically with the degree of polynomials. Thus, with the increase of the number of \( e_i \) in each \( \text{Stag}(w_j) \), the generation process of \( F_{i,j} \) will consume more time, which is also the major overhead of the proof generation time. The percentages of the computational overhead is dominated by the computation cost of \( acc(w_j) \), which completely hides the hashing cost and the tree construction cost in the setup stage. Given \( m=64 \) in Fig. 9, we can see the total setup time is about 170s when the number of \( n_w \) reaches 10,000. The total setup time which

### Table 1: Costs of Primitive Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP ( G_1/G_T )</td>
<td>0.55/0.94ms</td>
</tr>
<tr>
<td>Mul in ( G_q/G_T )</td>
<td>7( \mu )/0.09ms</td>
</tr>
<tr>
<td>SHA-256/Pairing</td>
<td>5( \mu )/1.41ms</td>
</tr>
<tr>
<td>Acc in ( G_1 ) (1000/1000/10,000 elements)</td>
<td>25.3/230/2628ms</td>
</tr>
<tr>
<td>Polyom. Mul in ( G_q ) (100/1000/10,000 coeff)</td>
<td>0.4/7.3/92.9ms</td>
</tr>
<tr>
<td>XGCD in ( Z_q )</td>
<td>8.4/509/108093ms</td>
</tr>
</tbody>
</table>

![Fig. 9: Setup time vs \( n_w \).](image1)

![Fig. 10: Proof generation time vs \( n_w \) where \( m=64 \).](image2)

B. Implementation

To demonstrate the efficiency of our scheme, we evaluate our scheme based on the benchmarks of [28] with the Intel Core i5 CPU 2.5GHz. During the implementation, the DCLXVI [29] is used for the bilinear pairing computations, Flint [30] for the modular arithmetic, and Crypto++ [31] for SHA-256 hash functions. DCLXVI employs a 256-bit BN elliptic curve and an asymmetric optimal ate pairing, offering bit-level security of 128 bits. We represent elements of \( G_1 \) with 768 bits using Jacobi coefficients, which yields faster operations. Elements in \( G_T \) are roughly twice as large as those of \( G_1 \). We summarize the cost of primitive operations in Table I [28]. We use the conjunction keyword query operation \( w_1 \) AND \( w_2 \) AND \( \cdots \) AND \( w_n \)" to demonstrate the efficiency of the proof generation and verification. The corresponding result is \( \{e_1, e_2, \cdots, e_k\} \).

1) Setup: To construct \( AT \), \( DO \) should compute \( acc(w_j) \), then compute the \( d(v) \) for each node of \( AT \). It is clear that, the total computation overhead of this phase only relates to the number of \( \text{ir}d \), \( e_i \) contained by each keyword \( w_i \) and the total nodes of \( AT \). Fig. 9 indicates the computation cost versus the number of index/documents \( n_w \) in each keyword when the number of keywords \( m \) is 9 and 64, respectively. We observe that the overhead increases almost linearly with the number of index/documents in each keyword. What causes the result is that the overhead is dominated by the computation cost of \( acc(w_j) \), which completely hides the hashing cost and the tree construction cost in the setup stage. Given \( m=64 \) in Fig. 9, we can see the total setup time is about 170s when the number of \( n_w \) reaches 10,000. The total setup time which

is required by each keyword in the accumulation operation increases linearly with the number of keywords. This is a one-time cost for the data owner. The additional storage overhead caused by \( AT \) is about 20KB for \( m=9, \epsilon=1/2 \) and 150KB for \( m=64, \epsilon=1/3 \).

2) Update: For a chosen \( \epsilon \), the computation overhead of each update process increases linearly with the number of keywords contained by the update index. We do not plot the update process. The update consumption for each process is approximately 2ms and 3ms for \( m=9 \) and \( m=64 \), respectively.

3) Proof Generation: Fig. 10 plots the proof generation time at the cloud server versus the number of index/documents \( n_w \) in each keyword. It shows that the proof generation overhead is influenced by the number of index/documents \( n_w \) in each keyword and the number of conjunction keywords \( n \). Given parameter \( m=64, \epsilon=1/3, n_w=10,000 \) and \( h=0.1n_w \), we can see that to generate the proof, it consumes about 430s and 1,740s for \( n=4 \) and \( n=16 \), respectively. The larger the number of conjunction keywords \( n \) is, the more time this proof generation phase takes. The reason leading to such result is that with the number of conjunction keywords increasing, the number of the set check process required to be executed grows correspondingly. It needs more exponentiation operations and polynomial expansion operations while the number of index \( n_w \) and \( h \) in each keyword are the same.
C. Extension performance

As shown in Fig. 10, the computation overhead of proof generation increases sharply with the number of \( n_w \) in each keyword, which may make our verification scheme unscalable. To deal with this problem, we propose our extended scheme. To demonstrate the efficiency of our extended scheme, we give the performance evaluation in Fig. 12, where \( n = 16 \) and \( m = 64 \). Here, we only use the extended scheme when \( n_w = 10,000 \) and we assume that each \( \text{Stag}(w_{i,j}) \) has 1000 encrypted indices. Fig. 12(a) plots the proof generation time of the basic scheme (1,740s) with the best and worst condition of the extended scheme (about 17s and 170s) when \( n_w = 10,000 \), respectively. From the comparison, the extended scheme is much more efficient than the basic scheme since the degree of polynomials in each group decreases, which makes the computation time of XGCD acceptable. Thus, the extended scheme is more practical. Notice that even in the worst condition, if we compute the proof in parallel, the time consumption is almost the same as the best situation. Fig. 12(b) shows the proof verification time of the basic scheme (3s) and the best and worst condition of the extended scheme (about 3s and 4.5s) when \( n_w = 10,000 \), respectively. The proof verification process should be executed according to the number of groups where the result distributes inside. However, each degree of polynomials becomes lower, which makes the accumulation process more efficient. From the comparison, even in the worst condition, the verification scheme is still practical.

VII. RELATED WORK

We introduce related work in three aspects: searchable encryption, query integrity verification, and query integrity verification on encrypted data.

**Searchable encryption**: Song et al. [32] explicitly considered the problem of searchable encryption and presented a scheme with search time that was linear with the size of the data collection for the first time. Their construction supports insertions/deletions of documents in a straightforward way. Curtmola et al. [21] gave the first index-based SSE constructions to achieve sublinear search time for SSE. A similar construction was proposed by Chase and Kamara [33], but with higher space complexity. Subsequently, Kamara et al. [3] introduced a dynamic scheme which was the first one with sublinear search time. However, it did not achieve forward privacy or revealed hashes of the unique keywords contained in the document during the update. Recently, Cash et al. [4] presented a SSE scheme supporting conjunction queries over static data. Based on Cash et al. [4] work, Jarecki et al. [2] proposed a scheme that allowed data owners to authorize third parties and to execute private information retrieval on the outsourced database. Faber et al. [8] extended the search capabilities of the system from [4] by supporting range queries, substring queries, wildcard queries, and so on. Moreover, they also extended their techniques to the more involved multi-client SSE scenarios studied in [2]. However, they [2, 4, 8] did not consider the update process and the integrity query verification comparing with our scheme. To support efficient update and preserve the privacy during the update process, Stefanov et al. [6], Cash et al. [7], and Naveed et al. [5] proposed their schemes. It is obvious that, all the proposed schemes for SSE do not consider the verification problem of the search result.

**Query integrity verification**: Li et al. [14] introduced an efficient implementation of Merkle hash tree authenticated \( B^+ \)-tree to audit the completeness of the query result, and demonstrated its superiority over signature chaining. By using signature aggregation, Pang et al. [13] proposed a scheme to verify the freshness, authenticity, and completeness of query answers from frequently updated databases that were hosted on untrusted servers. Li et al. [16] considered verifying queries
on a data stream with sliding windows via Merkle trees, hence the verifier’s space was proportional to the window size. Papadopoulos et al. [17] proposed a protocol to verify continuous query over streaming data, again requiring linear space on the verifier’s side in the worst case. Instead of using cryptographic primitives to verify the query result, Yi et al. [18] used algebraic and probabilistic techniques to compute a small synopsis on the true query result and to store a compact authentication synopsis that helped audit the result integrity. Papamanthou et al. [23] adopted the accumulation tree to verify the correctness of set operations. However, they did not consider PIR and the inefficiency of the accumulation tree with large datasets. To achieve public verification, Nath et al. [15] proposed DiSH, a small and efficient signature to verify outsourced grouped aggregation queries on streaming data. Azraoui et al. [9] used the well-established techniques of Cuckoo hashing, polynomial-based accumulators, and Merkle trees to publicly verify conjunctive keyword search in outsourced databases. However, unlike our solution, none of these solutions achieve public verification for encrypted data.

**Query integrity verification on encrypted data:** Kurosawa et al. [34] showed how to construct a (verifiable) SSE scheme that was universally composable (UC). While UC-security is a stronger notion of security, its construction requires linear search time. In addition, it did not consider the Boolean query on keywords. Zheng et al. [19] proposed a verifiable attribute-based keyword search scheme which can verify whether the cloud had faithfully executed the search operation by using a signature and Bloom filter. However, to enable general multiple keywords Boolean query, the query contains a number of keywords which are not known in advance and their preceding signature may not work. Apart from that, their scheme did not consider the update process and update needed to resign documents, which is time-consuming.

**VIII. CONCLUSION**

In this paper, we study the problem of verifying the freshness, authenticity, and completeness of the Boolean query result over the outsourced encrypted data. Based on OSPIR-OXT [2], we propose a publicly verifiable scheme by constructing the accumulation tree to achieve the query integrity verification while keeping privacy-preserving and efficiently practical. The security analysis shows that without protecting the access pattern, our scheme can keep the privacy-preserving of private information retrieval. The performance demonstrates our scheme is scalable.

**REFERENCES**


