Se-PKSE: Secure Public-Key Searchable Encryption for Cloud-Assisted Lightweight Platforms

Israt Jahan Mouri

Bangladesh University of Engineering and Technology

Muhammad Ridowan

Tiger IT, Bangladesh

Muhammad Abdullah Adnan (✉ abdullah.adnan@gmail.com)

Bangladesh University of Engineering and Technology <https://orcid.org/0000-0003-3219-9053>

Research Article

Keywords: Forward Privacy, Dynamic PKSE, Search Keyword Privacy, Cloud, Searchable Encryption, Lightweight Platforms, IoT, Ranked Search

Posted Date: August 25th, 2021

DOI: <https://doi.org/10.21203/rs.3.rs-796816/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

RESEARCH

Se-PKSE: Secure Public-Key Searchable Encryption for Cloud-Assisted Lightweight Platforms

Israt Jahan Mouri¹, Muhammad Ridowan² and Muhammad Abdullah Adnan^{1*}

Abstract

Since more and more data from lightweight platforms like IoT devices are being outsourced to the cloud, the need to ensure privacy while retaining data usability is important. Encrypting documents before uploading to the cloud, ensures privacy but reduces data usability. Searchable encryption, specially public-key searchable encryption (PKSE), allows secure keyword search in the cloud over encrypted documents uploaded from IoT devices. However, most existing PKSE schemes focus on returning all the files that match the queried keyword, which is not practical. To achieve a secure, practical, and efficient keyword search, we design a dynamic ranked PKSE framework over encrypted cloud data named *Secure Public-Key Searchable Encryption* (Se-PKSE). We leverage a partially homomorphically encrypted index tree structure that provides sub-linear ranked search capability and allows dynamic insertion/deletion of documents without the owner storing any document details. An interactive search mechanism is introduced between the user and the cloud to eliminate trapdoors from the search request to ensure search keyword privacy and forward privacy. Finally, we implement a prototype of Se-PKSE and test it in the Amazon EC2 for practicality using the RFC dataset. The comprehensive evaluation demonstrates that Se-PKSE is efficient and secure for practical deployment.

Keywords: Forward Privacy; Dynamic PKSE; Search Keyword Privacy; Cloud; Searchable Encryption; Lightweight Platforms; IoT; Ranked Search

Introduction

Cloud-based public-key searchable encryption (PKSE) schemes [1, 2, 3] enable lightweight platforms (limited CPU, memory, or storage) like sensors, mobile devices, wearable devices, drones, and other smart devices to encrypt (using the public key), outsource and dynamically maintain documents in the cloud. Simultaneously, these encrypted documents are searchable by an authorized user using the private key without letting the cloud know too much information.

The ranked searchable encryption is often used for retrieving partial search query results with relevance ranking instead of sending undifferentiated results to the users [4, 5]. A long line of research in PKSE focuses on returning all the files that match the queried keyword [2, 6, 7, 8]. This is not adequate for the large volume of information; that IoT devices can continuously produce.

Let us consider the following scenario. XYZ Inc. launches a new mobile application where people can log their symptoms. Cloud Inc. is providing a cloud back-end for the app where the symptom documents will be stored. The app uploads symptom documents in an encrypted form to keep keywords and documents secret. There could be many documents matching a specific keyword like *cough* or *fever*, so a doctor may want to perform a top-k search to retrieve the most relevant symptom documents associated with cough or fever. Therefore, XYZ Inc. maintains an index in the cloud to facilitate searching. Since the medical documents are confidential, XYZ Inc. needs to ensure that Cloud Inc. knows as little information about the documents as possible.

To achieve sub-linear ranked search efficiency, keyword balanced binary tree (KBB tree) based index structure is used for retrieving the documents from the cloud [9, 10, 11]. To avoid information leakage, KBB trees are generally encrypted before uploading. However, encryption disables the dynamic insertion/deletion of documents in the KBB tree. For this,

* Correspondence: abdullah.adnan@gmail.com

¹Department of Computer Science & Engineering, Bangladesh University of Engineering & Technology, Dhaka, Bangladesh

Full list of author information is available at the end of the article

in current KBB tree-based schemes, updating document collection requires the owner to update the index and send it to the cloud [9, 11] which is not suitable for lightweight devices. Some of the schemes use order-preserving encryption (OPE) to update index in the cloud [10]; however, this leaks ordering information to the cloud. This research aims to construct an encrypted KBB tree-based index structure, which will be constructed and maintained in the cloud without revealing ordering information while allowing dynamic addition and deletion on the encrypted tree without the document owner's interaction.

All practical searchable encryption (SE) schemes need to reveal some information during searches and updates called *leakage* to ensure efficiency. The leakages can be of different forms depending on the schemes' adversarial model. The adversary can use these leakages to perform different types of leakage abuse attacks [12, 13, 14, 15]. Stefanov et al. [16] proposed two security features to resist leakage-abuse attacks in dynamic schemes: (1) *Forward Private or Forward Secure (FS)* - newly uploaded documents should not match a previously queried keyword, and (2) *Backward Private* - search queries should not leak matching entries after they have been deleted. Forward privacy is now a must-have for any dynamic SE, especially to resist Zhang et al.'s [14] devastating file injection attack. Backward privacy is a less researched topic, and very few schemes support backward privacy with varying degrees of efficiency. We do not consider backward privacy, as in our scheme's search procedure, an 'honest but curious' cloud server will not be able to include deleted documents in the search. Instead, we focus on achieving a forward private, dynamic, and ranked PKSE scheme in the cloud supporting lightweight document producers with small static storage.

This paper proposes an encrypted modified version of *KBB tree*-based search scheme, which supports single-keyword ranked search and dynamic addition/deletion of documents from lightweight devices like IoT or smartphones. We formally named our PKSE framework as Secure Public-Key Searchable Encryption (Se-PKSE). In Se-PKSE, the document producers encrypt the documents and create an encrypted document policy (*partially homomorphically encrypted normalized term frequency (TF)*). Both of them are outsourced to the cloud. The cloud then generates an encrypted dynamic KBB tree-based index from the encrypted document policies. Intermediate nodes are constructed using partial homomorphic addition that preserves but does not leak ordering information. The use of partial homomorphic encryption instead of much slower fully homomorphic encryption, allows our document producers to have low computation power and storage.

We construct an interactive *trapdoor-less* search mechanism that requires cooperation between the cloud and the user. The authorized user can initiate a search request for a specific keyword to the cloud from its reasonably good device with high bandwidth, like a standard workstation in an office. It can retrieve the best matched (highest *TF* score) or top-k documents through the search. Our contribution can be summarized as follows:

- 1 We propose and implemented a ranked searchable encryption scheme in the cloud named Se-PKSE suitable for lightweight document producers. Document producers can encrypt using the public key (partial homomorphic encryption), upload, and then remove the documents. The encryption needs $O(N)$ cryptographic operations on a document insertion where N is the number of keywords in the dictionary. To the best of our knowledge, this is the first cloud-based PKSE for IoT that supports ranked top-k document retrieval.
- 2 An authorized user can perform an interactive single keyword search with the cloud without using trapdoors. The search requires $O(\log |F|)$ times communication between the user and the cloud, where F is the set of current documents in the cloud.
- 3 Our scheme minimizes index information leakage, resists cloud-based statistical attacks, and is forward private. We provide proof of these securities against 'honest-but-curious' adversaries.

We have implemented the Se-PKSE scheme using Java. Javallier, a Java library for Paillier partial homomorphic encryption, is used as an encryption framework. RFC [17] dataset is used as sample document set. For implementing the application in cloud, we have used Amazon EC2 cloud service's *t2.nano* (lightweight document producer), *t2.micro* (data user in standard workstation) and *t3a.xlarge* (powerful cloud server) instances. Our implementation achieves an average insertion time of 10ms and an average searching time of less than 3s on the whole dataset.

The rest of this paper is organized as follows. Section [Related Works](#) provides an overview of the related works. Section [Problem Formulation](#) presents the overview of Se-PKSE and security models. Next, Section [Preliminaries](#) introduces the preliminary concepts. Section [Se-PKSE Construction](#) presents Se-PKSE construction with leakages and security analysis. Section [Performance Evaluation](#) includes experimentation details and performance analysis. Finally, we conclude in Section [Conclusions](#).

Related Works

Boneh et al. [1] proposed the first public-key based SE scheme, which avoids interaction between docu-

ment producer and user. Many highly efficient symmetric searchable encryptions (SSE) [9, 16, 18] have also been proposed. However, they are not suitable for lightweight platforms where there is a risk of data encryption key leakage. Here, we review some researches, which is more relevant to our work.

Lightweight Platforms. Over the last decade, the use of lightweight IoT devices are on the rise. Due to these devices' resource limitation and cloud servers' on-demand access facility to applications and data from any device, many PKSE scheme's [2, 3, 6, 7, 8, 19] allow these devices to outsource their data in cloud servers in a privacy-preserving manner. Chen et al. [2] proposed a lightweight searchable public-key encryption with forward privacy. They have a dedicated certificate authority (CA) to support a lightweight document owner. Many schemes use dedicated certificate authority to support lightweight document producers. Recently, Zhang et al. [8], and Chen et al. [6] used the blockchain-based techniques instead of a certificate authority. However, all the schemes mentioned above focus on returning all the files that match the queried keyword. This is not adequate for the large volume of information the lightweight platforms continuously produce.

Ranked Search. Wang et al. [4, 5] first proposed the scheme of secure ranked keyword search over encrypted cloud data. The authors used an inverted index based structure to accommodate keyword searching. Cao et al. [20], and Sun et al. [21] also presented a multi-keyword ranked search scheme using an inverted index structure. However, these schemes require rebuilding the entire index to perform both keyword and document update operations.

Xia et al. [9] presented a secure multi-keyword ranked search scheme using a KBB tree-based index structure. The index is constructed using *TF-IDF* based relevance scores. They randomize the scores using Gaussian random matrices to ensure privacy. Smithamol and Sridhar [10] presented a secure, dynamic, and parallel search enabled conjunctive search (PECS) framework using a tree-based partitioned index structure (TPIS). Peng et al. [11] proposed a tree-based ranked multi-keyword search scheme in a multi-data owner model. In Peng's scheme, a tree-based index is constructed for each data owner, and the cloud server then merges these indexes to support the multi-data owner model. None of the ranked search schemes mentioned above support dynamic insertion/deletion of documents without the owner storing some document details. Subsequently, as an improvement, Kabir and Adnan's [22] work saves communication overhead by reducing the burden of dynamic insertion/deletion from data owners to the cloud. However, this scheme

reveals some term frequency & document frequency information to the cloud.

Forward Privacy. All practical SE leaks some information, which is called leakages to ensure efficiency. The leakages can be of different forms depending on the schemes' adversarial model. Islam et al. [12] first analyzed the effect of leakages in SE. Cash et al. [13], and Blackstone et al. [15] designed new leakage-abuse attacks and showed that even small leakages can cause serious security problems. Moreover, Zhang et al. [14] has shown that dynamic schemes are more susceptible to devastating file injection attack as the search keyword from trapdoors can be recovered by inserting only a few files. This adaptive attack has brought an emphasis on forward privacy. Forward privacy was first achieved by Stefanov et al. using oblivious RAM [16]. Over the years, many efficient forward private SSE schemes have been researched like Bost [23], Bost et al. [24], Etemad et al. [25], Sun et al. [26], Chamani et al. [27] etc. However, all the schemes mentioned above are based on symmetric-key cryptography. They generally suffer from key management and distribution problems and are not suitable for IoT platforms. Forward private PKSE scheme is still a less discussed topic. Some recent forward private PKSE schemes include Chen et al. [2], Zhang et al. [3], Chen et al. [6] etc. but none of the schemes support ranked search functionality. Table 1 presents a summary of the features of various searchable encryption schemes.

To the best of our knowledge, no other PKSE schemes for lightweight platforms provide sub-linear and dynamic ranked search functionality with forward privacy.

Table 1 Features of various searchable encryptions

	Ref	FS	PKSE	Sub-Linear	Ranked
IoT	Chen et al. [2]	✓	✓	✓	-
	Zhang et al. [8]	-	✓	-	-
	Chen et al. [6]	✓	✓	-	-
	Miao et al. [7]	-	✓	-	-
Ranked	Xia et al. [9]	-	-	✓	✓
	Smithamol et al. [10]	-	-	✓	✓
	Peng et al. [11]	-	-	✓	✓
	Kabir et al. [22]	-	-	✓	✓
	Se-PKSE	✓	✓	✓	✓

Problem Formulation

In this section, we formally define the Secure Public-Key Searchable Encryption(Se-PKSE) and its security model.

Overview of Se-PKSE

Our proposed Se-PKSE involves three different entities, as shown in Fig. 1:

Document Producers. Document producers are lightweight and independent (added/removed dynamically) devices like IoT or smartphones that generate continuous collection of documents $F = \{f_1, f_2, f_3, \dots\}$, to be securely outsourced to cloud. These documents can be mobile application logs, IoT devices' sensor information, or health organizations' medical reports. Encrypted document policies are created from these documents using the public key, outsourced to the cloud with encrypted documents. The documents can be deleted once they are uploaded to the cloud.

Cloud Server. The cloud server stores the encrypted documents and constructs a dynamic, searchable KBB tree-based index from the encrypted document policies. Only the authorized user can request documents, and the search operation is performed in collaboration with the user.

Data User. The data user is the authorized user who initiates the search and has the private key. He has reasonable bandwidth, CPU, and stable connectivity with the server. He collaborates with the cloud to perform a keyword-based single or top-k ranked search using the private key.

Formally, **Se-PKSE** is defined by the following algorithms:

Setup(λ): takes security parameter λ as input and outputs the necessary system parameters $\mathcal{P}$.

KeyGen($\mathcal{P}$): takes system parameters $\mathcal{P}$ as input and outputs public-private key pair (PubKey, PrvKey). The keys are distributed to document producers and user, respectively.

GenDocPolicy(f , PubKey): takes document f and public key PubKey and outputs encrypted document policy using PubKey.

BuildIndexTree(docPolicy, Tree): takes document policy docPolicy and existing index tree Tree as input and then docPolicy is added to the Tree.

Search(keyword, k, PrvKey, Tree): takes search keyword keyword, private key PrvKey, and index tree Tree as input. It is a two party algorithm, consists of **searchUser**(keyword, k, PrvKey) and **searchCloud**(Tree). It outputs top-k documents (documents with highest TF score) from the Tree.

Security Model

The document producers and the data users in our proposed scheme are trusted entities, except the cloud server. The cloud is termed as “*honest, but curious*” in our proposed model. Cloud honestly executes every assigned task; meanwhile, it is curious about the encrypted documents. The leakages of Se-PKSE can be defined as $\mathcal{L}_{Se-PKSE} = (\mathcal{L}_{Setup}, \mathcal{L}_{BuildIndexTree}, \mathcal{L}_{Search})$. The details of the leakages are discussed in section **Leakage**. Formally, we define the following security models for Se-PKSE. The security models also imply security guarantees against the outside adversaries, which have fewer capabilities than the cloud.

Security of Document Policy and Index-Tree

The document policy is encrypted by the document producers and decrypted only by the user. Cloud has only access to encrypted document policies. An adversary in the cloud may try to infer document-related information from the encrypted document policies. Our security definition of document policy follows the security notions of Boneh et al. [1], Curtmola et al. [28] and Chen et al. [29] but we replaced *trapdoors* with *keyword search procedure* and *ciphertexts* with *document policies*. It guarantees that no adversary can distinguish a document policy from another one. That is, the document policy does not reveal any information about the underlying keywords to any adversary. Precisely, we introduce a game, namely indistinguishability under adaptive chosen keyword attack (IND-PKSE-CKA), to capture the security of document policy. The security game for IND-PKSE-CKA is defined as,

Definition 1 *IND-PKSE-CKA is an interactive game between an adversary $\mathcal{A}$ and Challenger $\mathcal{C}$ as follows:*

Setup. $\mathcal{C}$ runs the **KeyGen**($\mathcal{P}$) algorithm and gives the PubKey to $\mathcal{A}$. $\mathcal{A}$ has full control over the KBB tree. That is $\mathcal{A}$ can insert/delete any docPolicy in the tree.

Test query-1. $\mathcal{A}$ can adaptively request the docPolicy of any document f_i or initiate and retrieve top k search results for any keyword w of its choice from $\mathcal{C}$.

Challenge. $\mathcal{A}$ sends $\mathcal{C}$ two documents f_0 and f_1 of its choice which it wishes to be challenged. $\mathcal{C}$ picks $b \leftarrow \{0, 1\}$ and generates,

$$\text{docPolicy}_b \leftarrow \text{genDocPolicy}(f_b, \text{PubKey}) \quad (1)$$

$\mathcal{C}$ sends docPolicy_b to $\mathcal{A}$.

Test query-2. $\mathcal{A}$ can perform additional computations in polynomial time.

Output. $\mathcal{A}$ outputs guess $b' \in \{0, 1\}$, if $b' = b$ or in other words if $\mathcal{A}$ can correctly guess whether it was given the document policy of f_0 or f_1 , $\mathcal{A}$ wins the game. The advantage that $\mathcal{A}$ wins the game is defined as $\text{Adv}_{\mathcal{A}}^{\text{IND-PKSE-CSKA}} = \Pr[b' = b] - \frac{1}{2}$.

Security of Search Keyword

To capture the search keyword's security in terms of indistinguishability, we introduce a security game, namely indistinguishability under chosen search keyword attack for public-key searchable encryption (IND-PKSE-CSKA). It guarantees that adversary can not identify search keywords even if they are repeated. We define the security game for IND-PKSE-CSKA as,

Definition 2 *IND-PKSE-CSKA is an interactive game between an adversary $\mathcal{A}$ and Challenger $\mathcal{C}$ as follows:*

Setup. $\mathcal{C}$ runs the $\text{KeyGen}(\mathcal{P})$ algorithm and gives the PubKey to $\mathcal{A}$. $\mathcal{A}$ has read-only access to the KBB tree. That is $\mathcal{A}$ can observe insert/delete/search process in the tree.

Test query-1. $\mathcal{A}$ can adaptively request the docPolicy of any document f_i or initiate and retrieve search results for any keyword w of its choice from $\mathcal{C}$.

Challenge. $\mathcal{A}$ sends $\mathcal{C}$ two keywords w_0 and w_1 of its choice which it wishes to be challenged and which have not been searched before. $\mathcal{C}$ picks $b \leftarrow \{0, 1\}$. $\mathcal{C}$ initiates search for w_b and send the search result (document policy) docPolicy_b to $\mathcal{A}$.

Test query-2. $\mathcal{A}$ can perform additional computations in polynomial time.

Output. $\mathcal{A}$ outputs guess $b' \in \{0, 1\}$, if $b' = b$ or in other words if $\mathcal{A}$ can correctly guess whether it was given the search results for w_0 or w_1 , $\mathcal{A}$ wins the game. The advantage that $\mathcal{A}$ wins the game is defined as $\text{Adv}_{\mathcal{A}}^{\text{IND-PKSE-CSKA}} = \Pr[b' = b] - \frac{1}{2}$.

Forward Privacy

Forward privacy ensures the previous search leakages cannot be exploited to infer information about the newly added documents [16]. In our scheme, the update operation is defined as deletion of the old document from the cloud and adding the updated document (details in **Dynamic KBB Tree Construction**). Therefore, we will only discuss the Forward Privacy of adding a document. Formally, we define the forward privacy game FS-PKSE of our scheme as,

Definition 3 *FS-PKSE is an interactive game between an adversary $\mathcal{A}$ and Challenger $\mathcal{C}$ as follows:*

Setup. $\mathcal{C}$ runs the $\text{KeyGen}(\mathcal{P})$ algorithm. $\mathcal{A}$ chooses two keywords w_0 and w_1 from the dictionary.

Test query-1. $\mathcal{A}$ can adaptively initiate and retrieve top k search results for any keyword w of its choice including w_0 and w_1 from $\mathcal{C}$.

Challenge. $\mathcal{C}$ picks $b \leftarrow \{0, 1\}$, add w_b to a document f_b and generates,

$$\text{docPolicy}_b \leftarrow \text{genDocPolicy}(f_b, \text{PubKey}) \quad (2)$$

$\mathcal{C}$ sends docPolicy_b to $\mathcal{A}$.

Test query-2. $\mathcal{A}$ can perform additional computations in polynomial time.

Output. $\mathcal{A}$ outputs guess $b' \in \{0, 1\}$, if $b' = b$, $\mathcal{A}$ wins the game. The advantage that $\mathcal{A}$ wins the game is defined as $\text{Adv}_{\mathcal{A}}^{\text{FS-PKSE}} = \Pr[b' = b] - \frac{1}{2}$.

Preliminaries

Encryption Techniques

Homomorphic Encryption

To construct the modified KBB index tree, we need to perform addition on encrypted data using *homomorphic encryption (HE)*. It is a form of encryption that allows any data to remain encrypted while being processed and generates a result in encrypted form, which matches the result of the operations as if they were performed on the plaintext [30]. It can be categorized into three types of schemes [31]. (1) **Fully Homomorphic Encryption (FHE)**: it supports unlimited number of arbitrary functions (e.g., searching, sorting, max, min, etc.), unlimited number of times. Example : Helib, libScarab, Microsoft SEAL. (2) **Some-what Homomorphic Encryption (SwHE)**: It supports all types (a logically complete set) of operations (e.g., addition and multiplication) but a limited number of times (e.g., limited circuit depth). Example: Sande-Tomas-Young, Boneh-Goh-Nissim, Ishai-Yuval-Paskin. (3) **Partially Homomorphic Encryption (PHE)**: It supports one type of operation, unlimited

number of times. Example: Goldwasser-Micali Cryptosystem, Okamoto-Uchiyama Cryptosystem, Paillier Cryptosystem, El-Gamal Cryptosystem.

FHE is still not practical to use in today's big data world as it is impractically slow [32]. It uses complex and computationally heavy operations. *SwHE* is not practical [31] as it allows operation for a limited number of times. Therefore, we propose to use *PHE*, which supports only a single operation, which in our case, is addition, an unlimited number of times. Our scheme is oblivious to the *PHE* chosen, as long as it supports the following features:

Public-Key Encryption: The encryption is an asymmetric key PHE, consists of a public-private key pair (*PubKey*, *PrvKey*). Distributed document producers have the risk of encryption key leakage. Therefore symmetric key encryption is not suitable for our scheme.

Additive Homomorphism: The encryption supports an addition operator $\boxplus$ such that for $Enc(a, PubKey) \rightarrow e_a$ and $Enc(b, PubKey) \rightarrow e_b$, $Dec(e_a \boxplus e_b, PrvKey) \rightarrow a + b$.

Subtractive Homomorphism (optional): The encryption supports a subtraction operator $\boxminus$ such that for $Enc(a, PubKey) \rightarrow e_a$ and $Enc(b, PubKey) \rightarrow e_b$, $Dec(e_a \boxminus e_b, PrvKey) \rightarrow a - b$. This is an optional feature for our scheme. If it exists, it can be used for optimization.

Indistinguishability under Chosen Ciphertext Attack (IND-CCA1): We assume that our chosen PHE scheme is IND-CCA1 secure, which is the highest security level for HE (due to their malleability). Though general IND-CCA1 security of PHE is an open question, many PHE like Paillier cryptosystem has proven to be IND-CCA1 secure [33, 34]. Although, we would like to point out that an Indistinguishability under Chosen Plaintext Attack (IND-CPA) secure PHE can also work in our scheme with fewer security guarantees by modifying Definition 1 and 2. The definition of IND-CCA1 is given below:

Definition 4 Setup. The challenger runs the algorithm $KeyGen(\mathcal{P})$ and generates the public-private key pair and gives the public key to the adversary and keeps the private key to itself.

Test query-1. The adversary can adaptively request the challenger to call the encryption or decryption oracle for its choice of plaintext/ciphertext as many times as he wants.

Challenge. The adversary sends the Challenger two distinct chosen plaintexts of the same length $message_0$ and $message_1$ of its choice, which it wishes to be challenged. The challenger picks $b \leftarrow \{0, 1\}$ uniformly at random and generates a ciphertext $C_b =$

$Enc(PubKey, message_b)$. The challenger sends C_b back to the adversary.

Test query-2. The adversary can perform additional computations in polynomial time.

Output Finally, the adversary outputs a guess for the value of b . If $guess = b$, the adversary wins.

Any scheme is IND-CCA1 secure if no adversary has a non-negligible advantage in winning the above game.

The adversary wins the above IND-CCA1 game if the advantage of the adversary is greater than,

$$Adv_{adversary}^{IND-CCA1} = Pr[guess = b] - \frac{1}{2}.$$

Notations

- $\mathbf{F}$, The plain text document collections, $\mathbf{F} = \{f_1, f_2, f_3, f_4 \dots\}$.
- $\mathbf{C}$, The encrypted ciphertext document collections, $\mathbf{C} = \{c_1, c_2, c_3, c_4 \dots\}$.
- $\mathbf{W}$, The dictionary is the unique set of keywords used in search, $\mathbf{W} = \{w_1, w_2, w_3, \dots, w_N\}$.
- $N = |\mathbf{W}|$ is the number of words in $\mathbf{W}$.

Vector Space Model Construction

Vector space model with *term frequency-inverse document frequency (TF-IDF)*, is a numerical statistic that indicates how important or relevant a word is to a document in a collection of documents [35]. There are many variations of the *TF-IDF* weighting scheme. We have used the definition from [35] in our scheme. *Term Frequency (TF)* indicates the weight of a keyword in a document. Suppose we have a collection of N documents, FR_{ij} is the frequency of keyword $w_i \in W$ in document f_j and $\max_k FR_{kj}$ is the maximum frequency of any keyword $w_k \in W$ in document f_j . Then, TF_{ij} , is defined as: $TF_{ij} = \frac{FR_{ij}}{\max_k FR_{kj}}$.

In our scheme, we normalized the TF scores by multiplying it with a constant value of α and rounding it to the nearest integer. α is the normalization factor, which is combined with rounding function $\lceil \cdot \rceil$ to make the TF based relevance score an integer in $[0 - \alpha]$ range.

Inverse Document Frequency IDF indicates how important a word is in a document collection or how much information the word provides. In the future, we plan to extend our framework to support a multi-keyword ranked search using the *TF-IDF* model with pre-computed *IDF* [9].

Keyword Balanced Binary Tree (KBB tree) Construction Xia et al. [9] first proposed the concept of a KBB tree. It is a dynamic tree-based index structure where each leaf node denotes a specific document, f_j . A vector *Data* with length N is stored in each node. In the leaf level, for each document f_j , i^{th} position of vector, $Data[i]$ denotes the normalized TF_{ij} score of keyword

w_i . The internal tree nodes are generated by taking the maximum of the pairwise leaf nodes from left to right.

$$Data_{parent}[i] \leftarrow$$

$$max(Data_{leftChild}[i], Data_{rightChild}[i]), i = 1, \dots, N$$

Xia et al. implement searching using a greedy depth-first search algorithm (GDFS). The user first creates a vector *trapdoor* for the searched keywords and sends it to the cloud. The relevance score *RScore* of any node in the index tree j is calculated using $RScore_j \leftarrow Data_j \cdot trapdoor$. The cloud uses the relevance scores to greedily select which nodes are accessed during GDFS and maintain top-k documents for the result. However, with this approach, the cloud can store the *trapdoor* and use it extensively without owner knowledge. The relevance scores are leaked to the cloud, which might be used in leakage abuse attacks. The updates also need the user to modify the KBB tree and re-upload it to the cloud.

Se-PKSE Construction

In this paper, we develop a secure PKSE search framework over encrypted cloud data that allows the cloud server to perform a ranked keyword search without knowing the document producer's sensitive information. As stated earlier, the proposed scheme consists of three entities: document producer, cloud server, and data user. Each entity plays a different role in the data outsourcing environment. The document producer encrypts the document and creates an encrypted document policy. Both of them are outsourced to the cloud. The cloud then generates an encrypted dynamic KBB tree-based index from the encrypted document policies according to the techniques discussed in [Proposed Modified KBB Tree](#). Our novel approach uses a partially homomorphic encrypted KBB tree-based index to mitigate the security concerns raised by a “*honest, but curious*” cloud server. A data user can initiate a search request for a specific keyword and retrieve the best matched (highest *TF* score) documents through an interactive search protocol.

Proposed Modified KBB Tree

In our proposed modified KBB tree, the leaf nodes store partially homomorphic encrypted normalized *TF* scores. The parent nodes are generated by the homomorphic addition of the left and the right child nodes.

$$Data_{parent}[i] \leftarrow Data_{leftChild}[i] \boxplus Data_{rightChild}[i] \\ i = 1, \dots, N$$

However, this parent node construction process nullifies relative order information between nodes which is required during searching. Because of that, it must be ensured that if the leaf node with the highest relevance score resides in the right subtree, the summation

of relevance scores in the right subtree must be larger than the summation of all leaves of the left subtree. To solve this, we define an encoder $b(): \mathbb{N}_0 \rightarrow \mathbb{N}_0$ to preserve relative order through the sum.

Theorem 1 *If the encoder $b()$ can be defined as:*

$$b(s) = M^s \quad (3)$$

Where,

- s is the normalized *TF* based relevance score
- $M = \lfloor \frac{D_{max}}{2} \rfloor + 1$, where D_{max} is the maximum number of documents the system can handle. This proof assumes that, $D_{max} > 1$.

Then, the encoder $b()$ has the following properties:

- $b(0)$ is the smallest output of b
- $b(s_{max}) > \sum b(j)$, here, s_{max} is the maximum relevance score in the tree and $j \in$ all relevance scores of the subtree where s_{max} does not reside

Proof For the first property,

$$b(0) = M^0 = 1 \quad (4)$$

which is smallest value of M^s where s is non-negative. This proves the first property.

For the second property, equation 3 can be rewritten as,

$$b(s) = M * M^{s-1} = M * b(s-1) > (M-1) * b(s-1) \quad (5)$$

Hence, $b(s) > (M-1) * b(s-1)$ ensures the second property $b(s_{max}) > \sum b(j)$ that if the maximum value leaf node resides on the right subtree of root, the sum of all leaf nodes of the left subtree won't be greater than the maximum value. $\square$

Fig. 2 depicts our searching in the proposed KBB tree with only single keyword ($N = 1$) for searching the most relevant document ($k = 1$). In Fig. 2(a), we have shown a simple KBB tree without encoding. In Fig. 2(b), the relevance scores are encoded using $b()$. In both case, the search starts at the root node and reaches the leaf node by following the path of the node with maximum value. The Theorem 1 enables us to use the same searching mechanism in both KBB trees by preserving relative ordering through homomorphic addition. The encoded relevance scores are encrypted to create document policies (details in [Encrypted Document and Document Policy Generation](#)) and the KBB tree is constructed in cloud using partial homomorphic addition (details in [Dynamic KBB Tree Construction](#)). The two properties of $b()$ ensures that the search procedures described in [Searching](#) give correct results.

Figure 2 Comparison of (a) original KBB tree and (b) our proposed KBB tree with cardinality of the dictionary $N = 1$. In (b) each leaf node's value is the encrypted normalized TF score applied to $b()$. The minimum value of M is 5 as the value of D_{max} in this example is 8. These figures also shows an example of the search process, in which the query vector Q searches for the document with maximum normalized TF score.

System Description

Setup

Setup() takes security parameter λ as input and outputs dictionary W with cardinality N , normalized factor α , the maximum number of documents the system can support D_{max} , and M . Though our system supports dynamic document addition functionality, yet a fixed dictionary W is used. Realistically, in IoT, mobile, or healthcare systems, the keyword space usually doesn't change much. Therefore, an adequately sized fixed dictionary will suffice. The dictionary can also be updated later to accommodate more keywords by doing a *Setup()* and migration but without changing public-private key pair. Care must be taken while choosing the value of α as a small value can cause unequal frequencies being tied. Thus potentially inaccurate results to a top-k query and large value can cause the system performance to hamper. Similarly, smaller M can introduce error in search, but larger M hampers performance. Though theoretically minimum value of $M = \lfloor \frac{D_{max}}{2} \rfloor + 1$ is needed, in a real dataset, keywords are more distributed, and hence values much less than the minimum M would work accurately.

Keygen

Keygen() takes system parameters as input and outputs public-private key pair ($PubKey$, $PrvKey$) for the underlying partial homomorphic encryption. The keys are distributed to document producers and user, respectively.

Encrypted Document and Document Policy Generation

Documents produced by the document producers can be encrypted using any standard file encryption technique. Therefore, the encryption process of documents is not the primary concern of this paper. Our proposed scheme mainly focuses on the construction of encrypted document policy. Formally, the encrypted document policy of a document f , is constructed using the following equation:

$$E = Enc(b(\lfloor \alpha * TF_1 \rfloor)), Enc(b(\lfloor \alpha * TF_2 \rfloor)), \dots, Enc(b(\lfloor \alpha * TF_N \rfloor)) \quad (6)$$

To generate a document policy,

- The frequency of each keyword in the dictionary is extracted from the document to calculate TF scores. $TF_i, i \in [1 - N]$ denotes TF score of i^{th} word in the dictionary for f .
- The TF score is then normalized by multiplying with α and rounded to an integer to be applied to encoder $b()$ of equation 5.
- The output of encoder $b()$ is then encrypted using the $PubKey$.

The formal construction process of generating a document policy is presented in Algorithm 1.

Dynamic KBB Tree Construction

In Section [Proposed Modified KBB Tree](#), we have briefly introduced our modified version of *KBB* index tree structure. The tree dynamically changes with the insertion or deletion of documents as the leaf node. A tree node can be defined as:

Algorithm 1: genDocPolicy

Data: Document f

Result: Encrypted f with encrypted document policy E to the cloud

foreach $w \in W$ **do**

    compute tf for w ;

$h \leftarrow b(\lfloor \alpha * tf \rfloor)$;

    add encrypted h to E ;

end

Send (Encrypted f , E) to cloud;

Figure 3 An example of the construction process of KBB tree based index. The leaf nodes (E_1 , E_2 , E_3 and E_4) represent 4 document policies and the internal parent nodes are generated by adding the *data* values of leaf nodes. Parent $E_{(1+2)}$ is constructed by homomorphically adding the *data* values of node E_1 and E_2 nodes. In a similar way $E_{(3+4)}$ and root node E is constructed.

Node $\leftarrow \langle \text{ID}, \text{Data}, \text{Left}, \text{Right}, \text{Parent} \rangle$
 where, ID denotes unique identifier for each tree node, Data is the encrypted document policy for leaf nodes, the homomorphic summation of its left and right child's data for other nodes. Left, Right and Parent indicates the left, right, and parent node, respectively. An example of our index tree is shown in Fig. 3.

Insertion: A leaf node in the index tree is introduced for each document in the collection with document ID and encrypted document policy as the Data. The internal tree nodes are computed by adding its left child and right child's data values according to the addition property of PHE. The formal construction process of the index tree is presented in Algorithm 2. In Algorithm 2, a new document policy is inserted in a free leaf node, and the parent nodes are updated accordingly. *freeNodes* represents all the free leaf nodes. If there are no more free leaf nodes, we increased the tree's height by duplicating the existing tree and merged both trees under a new root.

Deletion: The deletion of a node requires the document producer or the user to send the document identifier to the cloud, and the cloud deletes the node corresponding to that identifier. After the deletion of a node, the subsequent parent nodes are also updated to adjust the new changes in a similar fashion of data insertion in Algorithm 2. The data value of the deleted node gets subtracted from all the subsequent parent nodes. The formal deletion process of a node from the tree is presented in Algorithm 3.

Update: As we have a lightweight document producer so the document is not stored once they are uploaded to cloud. Therefore, one way to update in our system is that first deleting the old document and then reinserting the updated document.

Algorithm 2: buildIndexTree

Data: encrypted document c , doc policy $data$

Result: The Index tree

if There is no existing tree **then**

    Create a new leaf node u with $u.ID \leftarrow GenID()$,
      $u.left \leftarrow u.right \leftarrow null$ and $u.data \leftarrow null$;
     Insert u to *freeNodes*, u is the root of the index tree;

end

else if *freeNodes* is empty **then**

    Create a binary tree *newTree* of same size as the existing tree

foreach leaf of existing tree **do**

            Create a new leaf node u with
              $u.ID \leftarrow GenID()$,
              $u.left \leftarrow u.right \leftarrow null$ and
              $u.data \leftarrow null$;

            Insert u to *freeNodes* and *listOfNodes*;

end

while *listOfNodes* has more than 1 node **do**

while *listOfNodes* is not empty **do**

                Take and remove front two nodes

(u, v) from *listOfNodes*;

                Create a new parent node p ;

$p.data \leftarrow null$;

$p.left \leftarrow u, p.right \leftarrow v$;

$u.parent \leftarrow p, v.parent \leftarrow p$;

                Insert p to *tempNodeSet*;

end

            Replace *listOfNodes* with *tempNodeSet*
             and then clear *tempNodeSet*;

end

    Create a new node *newRoot* with existing tree's

    root as *newRoot.left*, new tree's root as

newRoot.right and existing tree's root's data as
     newRoot.data;

    Set *newRoot* as the new root and the combined
     tree as existing tree.

end

Take a free leaf node u from *freeNodes*. Store c with
 $u.ID$ in the cloud storage. Set $data$ in $u.data$;

while u is not null **do**

if $u.data = null$ **then**

$u.data \leftarrow data$;

end

else

$u.data \leftarrow u.data \boxplus data$;

end

$u \leftarrow u.parent$;

end

Algorithm 3: deleteNodeIndexTree

Data: The id of the document to be deleted, id
Result: The index tree
 Find the leaf node $node$ associated with id ;
 $data \leftarrow node.data$;
 $parent \leftarrow node.parent$;
 Remove $node$'s reference from its $parent$;
 Delete the document associated with id from cloud;
while $parent \neq null$ **do**
    $parent.data \leftarrow parent.data \boxminus data$;
    $parent \leftarrow parent.parent$;
end
 Add $node$ to $freeNodes$;

Searching

The searching in Se-PKSE is a recursive procedure on the index tree. Documents retrieval for a specific keyword from the cloud involves both the cloud and the user. A search request for a specific keyword is initiated from the user for single or top-k ranked documents (highest TF values for the searched keyword), and the request is passed to the cloud. The keyword is kept to the user so that no information related to the search query is transferred to the cloud. Cloud only provides subsequent access to specific index-tree nodes (selected by user) starting from the root node.

Single Document Search: In single document searching, at first, a search request for a specific keyword is initiated from the user. The keyword is kept to the user. The user asks the cloud to subtract the value of the right child node from the left child node of the root. The cloud passes the subtraction vector to the user. The user then decrypts the value that corresponds to the searched keyword using the private key, and if the decrypted value is *positive*, then the next node to be traversed in the index tree is the *left child node* else the *right child node*. The user then again repeats the same procedure for the next node. This procedure goes on until the cloud reaches the leaf node. Once the cloud reaches the leaf node, then it sends the *encrypted document* corresponding to that node to the user. Due to the particular KBB tree-based structure of the index, the number of communication between the user and cloud is kept to logarithmic. Therefore, this search mechanism involving the user and the cloud achieves sub-linear search efficiency with some round trip communication.

The formal searching procedure for the most relevant document on the user side is presented in Algorithm 5. For each node, the user takes *difference* as the homomorphic subtraction of the current node's left and right child's data from the cloud. The user then decrypts the part of *difference* associated with

Figure 4 An example of the search process that returns the most relevant document for the keyword “carol”. The cloud at first subtracts the value of $E_{(3+4)}$ from $E_{(1+2)}$ and then sends the result to the user. The user then decrypts the 3rd value as “carol” is the 3rd word in W . As the decrypted value is negative, the next node to be traversed in the tree is $E_{(3+4)}$. This decision is sent to the cloud, and this process goes on recursively until the user finds the node of interest. In this example as E_3 is the leaf node, the search process is terminated in this level, and the document that corresponds to E_3 is sent to the user.

that specific queried keyword using its $PrivKey$ and gets *decryptedValue*. The next node is left if $decryptedValue \geq 0$ and right if $decryptedValue < 0$. This process goes on recursively until the leaf node of interest is reached.

Algorithm 4: searchCloud

Data: id of the node to be traversed from user
Result: Subtracted value of child nodes or *null* if leaf node
 Find the leaf node $node$ associated with id ;
if $node.left = null$ and $node.right = null$ **then**
   return *null*;
end
else if $node.right.data = null$ **then**
   return $node.left.data$
end
else if $node.left.data = null$ **then**
   return $node.right.data$
end
else
   return $node.left.data \boxminus node.right.data$;
end

K Ranked Documents Search: The data user may initiate search process for the i^{th} keyword w_i to retrieve top-k documents. The search process is a recursive depth-first search (DFS) procedure upon the tree. The formal ranked search procedure is presented in Algorithm 6. We construct a result queue denoted as *priorityQueue*, whose element is defined as $(score, id)$.

Algorithm 5: searchUser

Data: SearchQuery for the i^{th} word where $w_i \in W$, communication protocol of *tree*

Result: Most relevant decrypted document f for w_i

Get root node's id id from the cloud;

while *TRUE* **do**

    Let, *difference* is the output of Algorithm 4 from the cloud for current node with id ;

if *difference* is null **then**

        Take encrypted document c with id from the cloud;

return $decrypt(c)$;

end

$decryptedValue \leftarrow decrypt(difference[i])$;

if $decryptedValue \geq 0$ **then**

        Get current node's left child's id from the cloud as id ;

end

else

        Get current node's right child's id from the cloud as id ;

end

end

Here, the *score* is the decrypted relevance score (i^{th} value of document policy) for the document with id . The *priorityQueue* stores the k accessed documents with the largest relevance scores to the query. The elements of the *priorityQueue* are ranked in descending order according to the *score*. It is updated during the search procedure.

From Theorem 1, we can see that the root's i^{th} value is the summation of all leaf nodes i^{th} value, which are various powers of M . That means we can find the relevance scores of all leaf nodes from the root node. Therefore, we can compute *minScore*, the minimum relevance score among the k documents, before running the DFS procedure just from the root node. By converting the root's value into a M base number and taking the highest significant digits until the sum is greater or equal to k , then the *minScore* is $M^{number\ of\ digits\ left}$. The DFS need not traverse the nodes which contain the relevance score less than *minScore*, and thus only parts of the tree nodes are accessed.

The ranked search procedure starts by the user taking the root node's value from the cloud and calculating the *minScore* for top- k ranked documents. Then, the user calculates the next node to be traversed after the root node. The user can calculate the values of the child nodes without accessing those nodes. We know that the root node is the summation of its child nodes. Furthermore, the user can obtain the subtraction vector of a node's child nodes from the cloud. The sum-

Algorithm 6: rankedSearchUser

Data: Searching for k documents for the i^{th} word $w_i \in W$, communication protocol with tree as *tree*

Result: Decrypted k ranked documents for w_i

Get root node's id id and data *data* from the cloud;

Decrypt *data*'s i^{th} element and store it in both *value* and *valueBackup*;

Convert *value* into a M base number

while *value* > 0 **do**

        Push (*value* mod M) to stack,

$value \leftarrow \lfloor \frac{value}{M} \rfloor$;

end

    Let, *digitSum* $\leftarrow 0$;

while *digitSum* $\leq k$ **do**

        pop from stack to *digitSum*;

end

Set *minScore* to $M^{stackSize}$;

Let, *priorityQueue* stores the k accessed documents with the largest relevance scores to the query and is always sorted by relevance score in descending order;

$dfsRanked(id, valueBackup)$;

return List of documents whose ids are in *priorityQueue* by retrieving them from the cloud;

---

def $dfsRanked(id, sum)$:

    Let, *difference* is the output of Algorithm 4 from the cloud for current node with id ;

if *difference* is null **then**

if *priorityQueue* has fewer elements than k **then**

            Insert (id, sum) to *priorityQueue*;

end

else

if *sum* is greater than smallest element of *priorityQueue* **then**

            Delete the element with smallest relevance score from *priorityQueue*;

            Insert (id, sum) to *priorityQueue*;

end

end

end

else

$sub \leftarrow decrypt(difference[i])$;

$leftData \leftarrow \frac{sum+sub}{2}$,

$rightData \leftarrow \frac{sum-sub}{2}$;

if $leftData \geq minScore$ **then**

$leftId \leftarrow$ current node's left child's id from cloud;

$dfsRanked(leftId, leftData)$;

end

if $rightData \geq minScore$ **then**

$rightId \leftarrow$ current node's right child's id from cloud;

$dfsRanked(rightId, rightData)$;

end

end

---

mation and subtraction of two nodes can be used to find the child nodes' values. These nodes are accessed if these values are higher than $minScore$. This process goes on recursively until the user finds k documents.

One thing worth mentioning in this search mechanism, it is assumed that the partial homomorphic encryption technique used here supports addition as well as subtraction. If homomorphic subtraction is not supported, both the child nodes of a specific parent node are sent to the user. The user then decrypts the vector's value that corresponds to the specific keyword from both the child nodes. These values are then subtracted from one another to decide the next node to be traversed in the index tree.

In both search procedures, all intermediate information is kept on the user side. The cloud only returns the node's data or homomorphic subtraction of data. Therefore, the cloud can serve multiple search requests in parallel.

Leakage

Se-PKSE reveals some information to the cloud server for efficiency, like all other practical SE schemes. The leakage of our scheme $\mathcal{L}_{Se-PKSE}$ can occur in three phases, **Setup**, **BuildIndexTree**, and **Search**. We have discussed the leakages of our construction below:

$\mathcal{L}_{Setup}$. The cloud performs partial homomorphic addition to generate the KBB index tree. Therefore, the cloud needs to know some parameters from the generated keys to perform the addition. Depending on the PHE chosen, the information can vary, though in most cases, this reveals the highest value, $hValue$ that can be encrypted with the generated public key. In Se-PKSE, The highest value to be encoded for a specific keyword in the dictionary is M^α . For performance, $hValue$ is close to M^α . The cloud can estimate M^α from $hValue$. Therefore, $\mathcal{L}_{Setup} = (M^\alpha)$.

$\mathcal{L}_{BuildIndexTree}$. During the **BuildIndexTree**, the cloud receives the encrypted document c , document policy and generates an id for the document. The cloud learns the size of c and dictionary size N (by observing the number of entries in the document policy). Therefore, $\mathcal{L}_{BuildIndexTree} = (N, |c|, id)$.

$\mathcal{L}_{Search}$. During the interactive **Search** procedure between the cloud and the user, the cloud can keep track of the search path through the KBB tree. The resulting documents for a queried keyword are retrieved by their ids from the cloud. Keeping track of the search path is not a leakage as if given k ids, the search path can be reconstructed because the path to any id from the root is unique. The only leakage is the response ids, which is also called the access pattern [12]. The leakage of access pattern in Se-PKSE implies that for same document collection, for same keyword, same ids are returned as a search result. Therefore, $\mathcal{L}_{Search} = (id_1, id_2, id_3, \dots, id_k)$.

Security Analysis

In Se-PKSE, the documents themselves can be encrypted with any standard file encryption technique. The security analysis of these standard encryption techniques is out of the scope of this research. Our goal is to prove the security models defined in Section **Security Model** against an "honest but curious" cloud server.

IND-PKSE-CKA: The formal security claim of the IND-PKSE-CKA game, defined in Definition 1, is given in Theorem 2.

Theorem 2 *If a probabilistic polynomial time (PPT) adversary $\mathcal{A}$ can break the **IND-PKSE-CKA** security of our scheme, there is a PPT adversary $\mathcal{B}$ who can break the **IND-CCA1** security of underlying PHE.*

Without loss of generality, let us assume, $\mathcal{A}$ is a subroutine of $\mathcal{B}$ and is transparent to the challenger $\mathcal{C}$. $\mathcal{A}$ is the adversary, and $\mathcal{B}$ is the challenger in the IND-PKSE-CKA game while $\mathcal{B}$ is the adversary and $\mathcal{C}$ is the challenger in the IND-CCA1 game.

Setup: $\mathcal{C}$ runs the $KeyGen(\mathcal{P})$ algorithm and gives the $PubKey$ to $\mathcal{B}$ and $\mathcal{B}$ sends the $PubKey$ to $\mathcal{A}$.

Test query-1:

Document policy generation/encryption $\mathcal{A}$ sends document f to $\mathcal{B}$. $\mathcal{B}$ generates the *plaintext document policy* vector, encrypts each element from $\mathcal{C}$, combines them to $docPolicy$ and send them back to $\mathcal{A}$.

Search/decryption $\mathcal{A}$ sends a keyword w to $\mathcal{B}$. $\mathcal{B}$ will perform the search procedure using $\mathcal{C}$ for decryption.

Challenge: $\mathcal{A}$ sends two documents f_0 and f_1 to $\mathcal{B}$. $\mathcal{B}$ generates two *plaintext document policy* vectors v_0 and v_1 for f_0 and f_1 respectively. For each position i , $\mathcal{B}$ sends $v_{0,i}$ and $v_{1,i}$ to $\mathcal{C}$. $\mathcal{C}$ picks a random value of $b \leftarrow \{0, 1\}$ and use the same b throughout the challenge. For all positions, $\mathcal{C}$ sends back $Enc(v_{b,i})$ to $\mathcal{B}$. $\mathcal{B}$ combines them to a $docPolicy$ and forwards it to $\mathcal{A}$.

Test query-2: $\mathcal{A}$ can perform additional computations in polynomial time.

Output: $\mathcal{A}$ outputs guess $b' \in \{0, 1\}$ and send to $\mathcal{B}$, and $\mathcal{B}$ will output b' as it's guess for b .

In the above process, from the view of challenger $\mathcal{C}$, $\mathcal{B}$ is an adversary who tries to break the IND-CCA1 security of the underlying PHE. From the view of $\mathcal{A}$, $\mathcal{B}$ is the challenger in the IND-PKSE-CKA game in our scheme. If $\mathcal{A}$ can break IND-PKSE-CKA, $\mathcal{B}$ can also break IND-CCA1. Therefore, the advantage of $\mathcal{B}$ breaking the IND-CCA1 security of underlying PHE is greater than or equal to $Adv_A^{IND-PKSE-CKA}$. Hence, the IND-PKSE-CKA of our scheme depends solely on the security of underlying PHE.

IND-PKSE-CSKA: The formal security claim of IND-PKSE-CSKA game, defined in Definition 2, is given in Theorem 3.

Theorem 3 *If a PPT adversary $\mathcal{A}$ can break the IND-PKSE-CSKA security of our scheme, $\mathcal{A}$ can also break the IND-PKSE-CKA security of our scheme.*

As shown in Section Leakage, the search leakage $\mathcal{L}_{Search}$ of our scheme is only limited to access pattern. Hahn and Kerschbaum [36] and later others [26, 37] shown that SE schemes can achieve security in terms of indistinguishability with access pattern leakage. Therefore, the search keyword's security depends on the security of the document policy, as discussed below.

From the perspective of $\mathcal{A}$, the challenge phases of the IND-PKSE-CSKA game and IND-PKSE-CKA game are similar. In the challenge phase of the IND-PKSE-CKA game, $\mathcal{A}$ receives a $docPolicy_b$ for a specific document f_b , and in the IND-PKSE-CSKA game, $\mathcal{A}$ receives a $docPolicy_b$, as a search result for a specific keyword w_b . However, the test phases of IND-PKSE-CSKA are a more restricted version of IND-PKSE-CKA. Therefore, $Adv_{\mathcal{A}}^{IND-PKSE-CSKA} \leq Adv_{\mathcal{A}}^{IND-PKSE-CKA}$.

Forward Privacy: The forward privacy of SePKSE, defined in Definition 3, is proved using Theorem 4.

Theorem 4 *Our scheme achieves forward privacy if a PPT adversary $\mathcal{A}$ has a negligible advantage of winning the IND-PKSE-CKA game.*

From the view of $\mathcal{A}$, $\mathcal{A}$ has fewer capabilities in FS-PKSEs' test query-1 and test query-2 phases than IND-PKSE-CKA. Theorem 2 proves that $\mathcal{A}$ cannot distinguish between document policies if the PHE is IND-CCA1 secure. The advantage that $\mathcal{A}$ wins the challenge depends on the security of document policies. Therefore, $Adv_{\mathcal{A}}^{FS-PKSE} \leq Adv_{\mathcal{A}}^{IND-PKSE-CKA}$.

Performance Evaluation

We have implemented the proposed scheme using Java language. Our entities (document producer, cloud server, and data user) are Java applications (jar) that connect through the REST protocol using Spring Framework. We have used Paillier partially homomorphic encryption scheme [38], to encrypt and decrypt the document policies generated from document producers. Javallier [39], a Java library for Paillier partially homomorphic encryption, is used as our encryption library.

The efficiency of the system is tested on the Request for Comments (RFC) dataset. The RFC dataset contains almost 8582 plain text documents with a total size of approximately 445 MB. RFC contains technical and organizational documents about the Internet, including the specifications and policy documents [17]. It is a widely used dataset in SE literature [4, 5, 9, 10, 11]. The dictionary W contains 1000 English words extracted from the RFC dataset.

For implementing the application in the cloud, we have used Amazon EC2 cloud service's *t3a.xlarge* (AMD EPYC 7000 series 2.5GHz processor with 4 core and 32GiB memory) instance as the *cloud server* and *t2.micro* (Intel Xeon 3.3 GHz processor with 1 core and 1GiB ram) instance for *data user*. And we took the lowest possible AWS instance *t2.nano* (Intel Xeon 3.3 GHz processor with 1 core and 0.5GiB ram) instance for *document producer*. The test includes: (1) the efficiency and storage requirements of single encrypted document policy generation, (2) the efficiency and storage requirements of dynamic index construction, and (3) the efficiency and accuracy of the search. We make the source code of our implementation open-source [40].

Encrypted Document Policy Generation

The encrypted document policy generation process includes two main steps:

- 1 *Construction of normalized vector:* The largest value for the normalized vector for a specific keyword in the dictionary is M^α .

- 2 *Encrypting the normalized vector with the public key of Pailliers partial homomorphic encryption:* The size of $(PubKey, PrvKey)$ pair depends on the value of $\alpha \log M$, minimum bits needed to encode M^α . For different combinations of M and α , the ciphertext size of Paillier is in the range of 144-704 bits.

There is an encrypted value for each keyword in the dictionary in the document policy. Hence, the size complexity of encrypted document policy is $O(N\alpha \log M)$. The time cost for encrypting a single document policy is proportional to the single encrypted policy's size. Fig. 5 shows how the average time for creating an encrypted document policy and the average size of encrypted document policy changes with different parameters.

Index Tree Construction

The time cost of index tree construction depends on:

- 1 *Number of documents:* The height of the tree expands as the number of documents increases. The insertion of a document as a leaf node only requires the subsequent parent nodes to be updated; therefore, only a single parent node is accessed at a particular level. Consequently, time changes logarithmically with the insertion of documents.
- 2 *Size of a encrypted document policy:* When a leaf node is inserted, a partial homomorphic addition is performed for each element in the document policy to update a parent node. The performance of the addition is dependent on the size (number

of bits) of the elements. Therefore, the time of inserting a single document policy is proportional to $N\alpha \log M$.

As the insertion and deletion operation is almost similar in the sense that insertion involves addition and deletion involves subtraction of encrypted values; therefore, the time complexity is also similar. The time complexity of generating index tree and deleting a node from the index tree is $O(\log |F| N\alpha \log M)$. Fig. 6 shows the insertion and deletion time.

On the other hand, the storage consumption of the index tree is determined by the size of the document collection, $|F|$. Specifically for $|F|$, the number of nodes in the KBB tree is $2|F| - 1$. In the KBB tree, every node stores a single encrypted document policy, and the document policy size is proportional to $N\alpha \log M$. Therefore, the storage requirement of index tree is $O(|F| N\alpha \log M)$. Fig. 7 shows the size of index tree with different parameters. From the graphs we can see that the storage consumption of the index tree is high. However, that is not a problem as storage is cheap in the cloud.

Search Efficiency

The time complexity of search mainly depends on:

- 1 *The height of the tree:* The height of the tree is $O(\log |F|)$. Therefore, the number of documents in the dataset has a small impact on the search.
- 2 *Network load:* The network load is proportional to the size of an encrypted document policy, which is $O(N\alpha \log M)$.

For the ranked search, the single ranked search's procedure is repeated k times. Therefore, complexity and network overload change accordingly. Fig. 8 shows the time cost of searching that returns a single document and k ranked documents with different parameters, respectively.

Search Accuracy

The search precision is defined by the rate of deviation in fetching correct documents for a specific keyword.

Figure 8 Average time cost for searching: single document (a) & (b) and k ranked documents (c) & (d). For (b) the collection of document is fixed, $F = 8582$.

Figure 9 A 3D graph is representing the error rate in search results for different α and M with fixed documents and dictionary, $N = 1000$.

In Theorem 1 we show that, the minimum value of M must be $\lfloor \frac{D_{max}}{2} \rfloor + 1$ to get accurate search result. For M , a value much smaller than the minimum value can be used in a practical scenario as in the real dataset, the keywords are more distributed. Fig. 9 illustrates that search precision is not affected by values of M ; different values of M always return the same document. However, precision is greatly affected by the value of α . Therefore, care must be taken while choosing the value of α as a smaller value of α can make two slightly different values of TF score normalized to the same value and thus returns incorrect results to a top- k query. Between Fig. 8 (c) and (d), we can see that decreasing M to 10% decrease the search time to 25-30%.

Performance Comparison

We compare the performance of Se-PKSE with Xia et al. [9] and Peng et al. [11]. We are comparing our

Figure 10 Average time for (a) single keyword search for different K and (b) score calculation & tree generation for different F in different ranked search schemes.

scheme with two SSE schemes instead of other IoT supported PKSE because no other IoT supported PKSE supports top- k ranked search. Fig. 10 shows that Xia et al.'s [9] scheme performs better than ours; however, updating document collection requires the owner to store the unencrypted index and send updates to the cloud. Storing a copy of the index tree is not feasible for IoT devices. Peng et al.'s [11] scheme is not efficient like Xia et al. as this scheme supports the multi-owner with different keys, and each data owner creates an index tree in his part, which is then merged in the cloud. SSE schemes are fast due to their faster symmetric-key encryption; however, they are not suitable for lightweight platforms as there is a risk of data encryption key leakage. The mentioned schemes also do not support forward security. Our goal is to make a PKSE for IoT, which will support top- k ranked search with forward security without being impractically slow. We think Se-PKSE is on the right path for implementing SE in the cloud on encrypted data from lightweight platforms.

Conclusions

This paper proposes a secure, forward private, efficient, and dynamic PKSE search framework over encrypted cloud data that supports single-keyword ranked search. We analyze the security of encrypted document policies, KBB based index tree, and search keywords. We have implemented our system in the Amazon EC2 cloud, and extensive experimental results demonstrate our solution's efficiency and scalability.

The scope of future research includes support of multi-user authentication using different private keys, and efficient multi-keyword ranked search with the TF - IDF model. In addition, Se-PKSE's encrypted KBB tree's path traversal during search has similarities with private decision tree evaluation. Therefore, further research can be done to use private decision tree evaluation techniques like [41] in Se-PKSE to increase efficiency.

Our scheme mainly considers the security challenges associated with an “honest but curious” cloud server, but privacy under a malicious cloud server should also be researched. Especially under a malicious cloud, backward privacy is required to resist leakage abuse attacks. Furthermore, some of the lightweight devices will likely be malicious in a distributed network for their diversity. These security issues can be incorporated in future researches.

Acknowledgements

The authors would like to thank all anonymous reviewers for their invaluable comments.

Abbreviations

PKSE: Public-key Searchable Encryption; KBB: Keyword Balanced Binary; OPE: Order Preserving Encryption; SE: Searchable Encryption; FS: Forward Secure; TF: Term Frequency; SSE: Symmetric Searchable Encryption; CA: Certificate Authority; PECS: Parallel search Enabled Conjunctive Search; TPIS: Tree-based Partitioned Index Structure; HE: Homomorphic Encryption; FHE: Fully Homomorphic Encryption; SwHE: Somewhat Homomorphic Encryption; PHE: Partially Homomorphic Encryption; TF-IDF: Term Frequency-Inverse Document Frequency; IND-CCA1: Indistinguishability under Chosen Ciphertext Attack; IND-CPA: Indistinguishability under Chosen Plaintext Attack; AWS: Amazon Web Services; RFC: Request for Comments

Availability of data and materials

The experiments are done on the Request for Comments (RFC) dataset [17] which at the time of the experiment contained 8582 plain text documents with a total size of approximately 445 MB. In addition, the implementation is shared publicly at [40].

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

The research work is a part of Israt Jahan Mouri master's work, which was conducted under the supervision and guidance of Muhammad Abdullah Adnan. Israt Jahan Mouri is the primary author of this research paper and the system architect and programmer of Se-PKSE. Muhammad Ridwan has contributed to the AWS deployment, management, and experimentation running. Muhammad Abdullah Adnan supervised and reviewed the associated experiments and contributed to the paper's literature review and general organization. All authors take part in the discussion, read and approved the final manuscript.

Authors' information

Israt Jahan Mouri has received her M.Sc. degree from Bangladesh University of Engineering and Technology and B.Sc. from the Military Institute of Science and Technology. Her research interests include data security in the cloud and searchable encryption. She is currently a Senior Lecturer at Stamford University Bangladesh.

Muhammad Ridwan has received his B.Sc from the University of Dhaka. His research interests include cloud computing, big data, and designing scalable and secure systems. He is currently leading development teams on highly scalable and cloud-native systems as Solution Architect in TigerIT Bangladesh Ltd. He has worked on several projects for the governments of Bangladesh, Nepal, Tajikistan, and Moldova.

Muhammad Abdullah Adnan is an Associate Professor in the Department of CSE at BUET. He received MSc Engg. and BSc Engg. degrees in Computer Science and Engineering from Bangladesh University of Engineering and Technology in 2008 and 2006, respectively. He completed his PhD from the Department of Computer Science and Engineering at the University of California San Diego (UCSD), the USA, in 2015. His research areas cover a wide range of topics in large scale systems that include Cloud Computing, Mobile Computing, BIG DATA Analytics, Data Science, Machine Learning for Large Scale Systems, Natural Language Processing, Green Computing, Data Center Networking, Smart Grid, and Internet-of-Things. Dr. Adnan has received many research awards and

grants from both scientific communities like Heidelberg Laureate Forum, Germany, North American Association of Computational Linguistics, USA and industries like Samsung, Korea, Amazon, USA and Government of Bangladesh. Dr. Adnan's research papers have been published in top-ranked journals and conferences such as the ACM/IEEE Transactions, Elsevier BigData Research, SIGKDD, EMNLP, WSDM, ICDE, IEEE Cloud, NAAAL, ICASSP, IEEE BigData, etc. Dr. Adnan is the lead of the Cloud Analytics Research Group at BUET.

Author details

¹Department of Computer Science & Engineering, Bangladesh University of Engineering & Technology, Dhaka, Bangladesh. ²TigerIT Bangladesh Ltd., Banani, Dhaka, Bangladesh.

References

1. Boneh, D., Di Crescenzo, G., Ostrovsky, R., Persiano, G.: Public key encryption with keyword search. In: Cachin, C., Camenisch, J.L. (eds.) *Advances in Cryptology - EUROCRYPT 2004*, pp. 506–522. Springer, Berlin, Heidelberg (2004)
2. Chen, B., Wu, L., Kumar, N., Choo, K.-K.R., He, D.: Lightweight searchable public-key encryption with forward privacy over iiot outsourced data. *IEEE Transactions on Emerging Topics in Computing* (2019)
3. Zhang, X., Xu, C., Wang, H., Zhang, Y., Wang, S.: Fs-peks: Lattice-based forward secure public-key encryption with keyword search for cloud-assisted industrial internet of things. *IEEE Transactions on Dependable and Secure Computing* (2019)
4. Wang, C., Cao, N., Li, J., Ren, K., Lou, W.: Secure ranked keyword search over encrypted cloud data. In: *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*. ICDCS '10, pp. 253–262. IEEE Computer Society, USA (2010). doi:[10.1109/ICDCS.2010.34](https://doi.org/10.1109/ICDCS.2010.34). <https://doi.org/10.1109/ICDCS.2010.34>
5. Wang, C., Cao, N., Ren, K., Lou, W.: Enabling secure and efficient ranked keyword search over outsourced cloud data. *IEEE Transactions on parallel and distributed systems* **23**(8), 1467–1479 (2011)
6. Chen, B., Wu, L., Wang, H., Zhou, L., He, D.: A blockchain-based searchable public-key encryption with forward and backward privacy for cloud-assisted vehicular social networks. *IEEE Transactions on Vehicular Technology* **69**(6), 5813–5825 (2020)
7. Miao, Y., Ma, J., Liu, X., Wang, J., Li, H., Li, H.: Lightweight fine-grained search over encrypted data in fog computing. *IEEE Transactions on Services Computing* **12**(5), 772–785 (2019)
8. Zhang, Y., Xu, C., Ni, J., Li, H., Shen, X.S.: Blockchain-assisted public-key encryption with keyword search against keyword guessing attacks for cloud storage. *IEEE Transactions on Cloud Computing* (2019)
9. Xia, Z., Wang, X., Sun, X., Wang, Q.: A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE transactions on parallel and distributed systems* **27**(2), 340–352 (2015)
10. Smithamol, M.B., Sridhar, R.: Pecs: Privacy enhanced conjunctive search over encrypted data in the cloud supporting parallel search. *Computer Communications* **126**, 50–63 (2018)
11. Peng, T., Lin, Y., Yao, X., Zhang, W.: An efficient ranked multi-keyword search for multiple data owners over encrypted cloud data. *IEEE Access* **6**, 21924–21933 (2018)
12. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: *NDSS*, vol. 20, p. 12. The Internet Society, Virginia, United States (2012). Citeseer
13. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 668–679 (2015)
14. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: *25th USENIX Security Symposium (USENIX Security 16)*, pp. 707–720. USENIX Association, Austin, TX (2016). <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang>
15. Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. In: *Proceedings 2020 Network and Distributed System*

- Security Symposium (2020)
16. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS, vol. 71, pp. 72–75. The Internet Society, Virginia, United States (2014)
  17. I.E.T.F.: Request for Comments (RFC). <https://www.rfc-editor.org/retrieve/bulk/>. [Online; accessed 02-February-2020] (2020)
  18. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of the 2000 IEEE Symposium on Security and Privacy. SP '00, p. 44. IEEE Computer Society, USA (2000)
  19. Xu, P., He, S., Wang, W., Susilo, W., Jin, H.: Lightweight searchable public-key encryption for cloud-assisted wireless sensor networks. *IEEE Transactions on Industrial Informatics* **14**(8), 3712–3723 (2018)
  20. Cao, N., Wang, C., Li, M., Ren, K., Lou, W.: Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on parallel and distributed systems* **25**(1), 222–233 (2013)
  21. Sun, W., Wang, B., Cao, N., Li, M., Lou, W., Hou, Y.T., Li, H.: Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. *IEEE Transactions on Parallel and Distributed Systems* **25**(11), 3025–3035 (2013)
  22. Kabir, T., Adnan, M.A.: A dynamic searchable encryption scheme for secure cloud server operation reserving multi-keyword ranked search. In: 2017 4th International Conference on Networking, Systems and Security (NSysS), pp. 1–9. ICPS, Dhaka, Bangladesh (2017). IEEE
  23. Bost, R.: Σ ϕ ρ σ : Forward secure searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16, pp. 1143–1154. ACM, New York, NY, USA (2016). doi:[10.1145/2976749.2978303](https://doi.org/10.1145/2976749.2978303). <http://doi.acm.org/10.1145/2976749.2978303>
  24. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1465–1482 (2017)
  25. Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. *Proceedings on Privacy Enhancing Technologies* **2018**(1), 5–20 (2018)
  26. Sun, S.-F., Yuan, X., Liu, J.K., Steinfeld, R., Sakzad, A., Vo, V., Nepal, S.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 763–780 (2018)
  27. Chamani, J.G., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: CCS 2018 : The 25th ACM Conference on Computer and Communications Security, pp. 1038–1055 (2018)
  28. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. CCS '06, pp. 79–88. Association for Computing Machinery, New York, NY, USA (2006). doi:[10.1145/1180405.1180417](https://doi.org/10.1145/1180405.1180417). <https://doi.org/10.1145/1180405.1180417>
  29. Chen, R., Mu, Y., Yang, G., Guo, F., Wang, X.: A new general framework for secure public key encryption with keyword search. In: Australasian Conference on Information Security and Privacy, pp. 59–76 (2015). Springer
  30. Rivest, R.L., Adleman, L., Dertouzos, M.L., *et al.*: On data banks and privacy homomorphisms. *Foundations of secure computation* **4**(11), 169–180 (1978)
  31. Acar, A., Aksu, H., Uluagac, A.S., Conti, M.: A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)* **51**(4), 79 (2018)
  32. Chirgwin, R.: IBM's homomorphic encryption accelerated to run 75 times faster. https://www.theregister.com/2018/03/08/ibm_faster_homomorphic_encryption/. [Online; accessed 05-January-2020] (2018)
  33. Lipmaa, H.: On the cca1-security of elgamal and damgård's elgamal. In: International Conference on Information Security and Cryptology, pp. 18–35 (2010). Springer
  34. Armknecht, F., Katzenbeisser, S., Peter, A.: Group homomorphic encryption: characterizations, impossibility results, and applications. *Designs, codes and cryptography* **67**(2), 209–232 (2013)
  35. Rajaraman, A., Ullman, J.D.: Data Mining, pp. 1–17. Cambridge University Press, Cambridge, England (2011). doi:[10.1017/CBO9781139058452.002](https://doi.org/10.1017/CBO9781139058452.002)
  36. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 310–320 (2014)
  37. Kim, K.S., Kim, M., Lee, D., Park, J.H., Kim, W.-H.: Forward secure dynamic searchable symmetric encryption with efficient updates. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1449–1463. ACM, New York, NY, USA (2017). ACM
  38. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques. EUROCRYPT'99, pp. 223–238. Springer, Berlin, Heidelberg (1999). <http://dl.acm.org/citation.cfm?id=1756123.1756146>
  39. Analytics, N.: javallier. <https://github.com/n1analytics/javallier>. [Online; accessed 01-January-2020] (2017)
  40. pkse-searcher: secure-pkse. <https://github.com/pkse-searcher/secure-pkse>. [Online; accessed 01-August-2021]
  41. Kiss, Á., Naderpour, M., Liu, J., Asokan, N., Schneider, T.: Sok: Modular and efficient private decision tree evaluation. *Proceedings on Privacy Enhancing Technologies* **2019**(2), 187–208 (2019)