RS-PKE: Ranked Searchable Public-Key Encryption for Cloud-Assisted Lightweight Platforms

Anonymous Authors

ABSTRACT

Since more and more data from lightweight platforms like IoT devices or mobile apps are being outsourced to the cloud, the need to ensure privacy while retaining data usability is essential. In this paper, we design a framework where lightweight platforms like IoT devices can encrypt documents and generate document indexes using the public key before uploading the document to the cloud, and an admin can search and retrieve the top-k most relevant documents that match a specific keyword using the private key. In most existing searchable encryption that supports IoT, all the documents that match a queried keyword are returned to the admin. This is not practical as IoT devices continuously upload data. We formally name our framework *Ranked Searchable Public-Key Encryption* (RS-PKE). We also implement a prototype of RS-PKE and test it in the Amazon EC2 cloud using the RFC dataset. The comprehensive evaluation demonstrates that RS-PKE is efficient and secure for practical deployment.

CCS CONCEPTS

• **Security and privacy** → **Management and querying of encrypted data**; • **Information systems** → Searching with auxiliary databases.

KEYWORDS

Forward Privacy, Dynamic PKE, Search Keyword Privacy, Cloud Security, Searchable Encryption, Lightweight Platforms, IoT, Ranked Search

ACM Reference Format:

Anonymous Authors. 2018. RS-PKE: Ranked Searchable Public-Key Encryption for Cloud-Assisted Lightweight Platforms. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Suppose 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. Therefore, thousands to million documents can match a specific keyword like *cough* or *fever*, so a doctor may want to analyze top 10 or top 20 most relevant symptom documents associated with *cough* or *fever*. Since the medical documents are confidential, XYZ Inc. needs to ensure that Cloud Inc. knows as little information about the documents as possible.

Cloud-based searchable public-key encryption schemes (SPKE) [4, 10, 34] enable lightweight platforms (limited CPU, memory, or storage) like sensors, mobile devices, wearable devices, drones, and other smart devices to encrypt the document, generate document index (using the public key), outsource and dynamically maintain documents and document indexes in the cloud. Simultaneously, these encrypted documents are searchable by an admin using the private key without letting the cloud know too much information. A long line of research focuses on returning all the files that match the queried keyword [10, 11, 22, 36]. This is not adequate for the large volume of information; that IoT devices can continuously produce. The ranked searchable encryption is used for retrieving partial search query results with relevance ranking instead of sending undifferentiated results to the owners [31, 32]. In this paper, we focus on returning the top-k most relevant documents in sub-linear time. To the best of our knowledge, no other work supports retrieving top-k relevant Public-key encrypted documents from the cloud.

In this paper, we use an index structure based on keyword balanced binary tree (KBB tree) [24, 26, 33] for ranked search with sub-linear efficiency. In current KBB tree-based schemes [24, 33], updating document collection requires the owner to update the index and send it to the cloud. For lightweight devices with limited bandwidth and storage, this is not suitable. [26] uses order-preserving encryption (OPE) to update index in the cloud; however, this leaks ordering information to the cloud. We aim 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.

We formally named our framework as Ranked Searchable Public-Key Encryption (RS-PKE). In RS-PKE, the document producers (lightweight devices) encrypt the documents (using any standard encryption) and create an encrypted document index (*partially homomorphically encrypted* normalized term frequency (TF) scores). Both of them are outsourced to the cloud. The cloud then generates an encrypted dynamic KBB tree-based index from the encrypted document indexes. 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 faster update in the cloud.

We construct an interactive search mechanism that requires cooperation between the cloud and the admin. The admin can initiate a search request for a specific keyword to the cloud from its reasonably good device with high bandwidth, like a standard workstation

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

in an office. It can retrieve the best matched (highest TF score) or top-k documents through the search.

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 [3, 8, 17, 35]. Stefanov et al. [28] 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 [35] devastating file injection attack. However, 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 SPKE scheme in the cloud supporting lightweight document producers with small static storage.

Our contribution can be summarized as follows:

(1) We propose and implement a ranked searchable encryption scheme in the cloud named RS-PKE, suitable for lightweight document producers. Document producers can encrypt the documents, generate document indexes, upload these in the cloud, and remove the documents. The document index generation needs $O(N)$ cryptographic operations where N is the number of keywords in the dictionary.
(2) An admin can perform an interactive single keyword search with the cloud without using trapdoors. The search requires $O(\log |F|)$ times communication between the admin 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 RS-PKE scheme using Java. Javallier, a Java library for Paillier partial homomorphic encryption, is used as an encryption framework. RFC [16] 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* (admin 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 2 provides an overview of the related works. Section 3 presents the overview of RS-PKE and security models. Next, Section 4 introduces the preliminary concepts. Section 5 presents RS-PKE construction with leakages and security analysis. Section 6 includes experimentation details and performance analysis. Finally, we conclude in Section 7.

2 RELATED WORKS

Boneh et al. [4] proposed the first public-key based SE scheme, which avoids interaction between document producer and admin. There are many highly efficient symmetric searchable encryptions (SSE) [27, 28, 33]. However, they are not suitable for lightweight platforms where there is a risk of 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 has been on the rise. Due to these devices' resource limitation and cloud servers' on-demand access facility to applications and data from any device, many SPKE scheme's [10, 11, 22, 34, 36] allow these devices to outsource their data in cloud servers in a privacy-preserving manner. Chen et al. [10] 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. [36], and Chen et al. [11] 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. [31, 32] 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. [7], and Sun et al. [30] 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. [33] 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. [26] presented a secure, dynamic, and parallel search enabled conjunctive search (PECS) framework using a tree-based partitioned index structure (TPIS). Peng et al. [24] 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 [18] 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 schemes leak some information called leakages to ensure efficiency. The leakages can be of different forms depending on the schemes' adversarial model. Islam et al. [17] first analyzed the effect of leakages in SE. Cash et al. [8], and Blackstone et al. [3] designed new leakage-abuse attacks and showed that even small leakages can cause serious security problems. Moreover, Zhang et al. [35] 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 [28]. Over the years, many efficient forward private SSE schemes have been researched like Bost [5], Bost et al. [6], Etemad et al. [14], Sun et al. [29], Chamani et al. [9] 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 SPKE scheme is still a less discussed topic. Some recent forward private SPKE schemes include Chen et al.[10], Zhang et al.[34], Chen et al. [11] 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 SPKE schemes for lightweight platforms provide sub-linear and dynamic ranked search functionality with forward privacy.

Table 1: Features of various searchable encryptions

Ref		FS	SPKE	Sub-Linear	Ranked
IoT	Chen et al.[10]	✓	✓	✓	-
	Zhang et al.[36]	-	✓	-	-
	Chen et al.[11]	✓	✓	-	-
	Miao et al.[22]	-	✓	-	-
Ranked	Xia et al.[33]	-	-	✓	✓
	Smithamol et al.[26]	-	-	✓	✓
	Peng et al.[24]	-	-	✓	✓
	Kabir et al.[18]	-	-	✓	✓
RS-PKE		✓	✓	✓	✓

3 PROBLEM FORMULATION

In this section, we formally define the Ranked Searchable Public-Key Encryption (RS-PKE) and its security model.

3.1 Overview of RS-PKE

RS-PKE 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 indexes 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 indexes.

Admin. The admin initiates the search and has the private key. He has reasonable bandwidth, CPU, and stable connectivity with the cloud. He collaborates with the cloud to perform a keyword-based single or top-k ranked search using the private key.

Formally, **RS-PKE** is defined by the following algorithms:

`Setup`(λ): takes security parameter λ as input and outputs the necessary system parameters $\mathcal{P}$. Admin executes it at beginning.

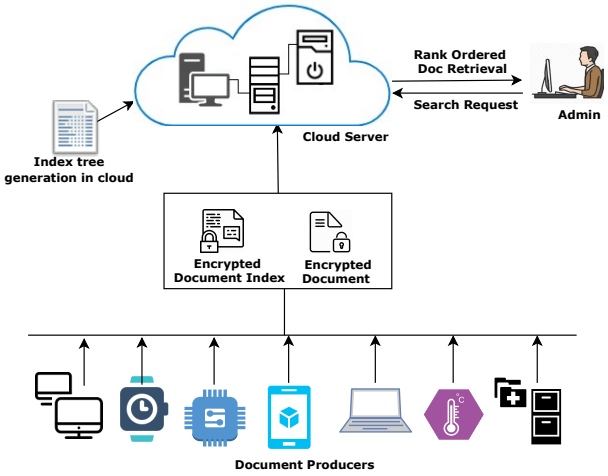

Figure 1: RS-PKE Model

`KeyGen`($\mathcal{P}$): takes system parameters $\mathcal{P}$ as input and outputs public-private key pair (`PubKey, PrvKey`). Admin distributes the public key to document producers and keep private key for search.

`GenDocIndex(f, PubKey)`: Document producer take document f and public key `PubKey` and outputs encrypted document index. This executes for each document to construct document index.

`BuildIndexTree(docIndex, Tree)`: The cloud takes encrypted document index `docIndex` and existing index tree `Tree` as input and then insert `docIndex` to `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 `searchAdmin(keyword, k, PrvKey)` and `searchCloud(Tree)`. It outputs top-k documents (documents with highest TF score) from the `Tree`. The cloud runs *BuildIndexTree* and *searchCloud*, and the admin runs *searchAdmin*.

3.2 Security Model

The document producers and the admin 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 and search keywords. The leakages of RS-PKE can be defined as $\mathcal{L}_{RS-PKE} = (\mathcal{L}_{Setup}, \mathcal{L}_{BuildIndexTree}, \mathcal{L}_{Search})$. The details of the leakages are discussed in section 5.3. Formally, we define the following security models for RS-PKE. The security models also imply security guarantees against the outside adversaries, which have fewer capabilities than the cloud.

3.2.1 Security of Document Index and Index-Tree. An adversary in the cloud may try to infer document related information from the index tree. The index tree is constructed from the document indexes. The security of the index tree depends on the document indexes. We introduce a game, namely indistinguishability under adaptive chosen keyword attack (IND-SPKE-CKA), to capture the security of document index. Our security definition of document index follows the security notions of Boneh et al.[4], Curtmola et al.[13] and Chen et al.[12] but we replace *trapdoors* with `Search(keyword,`

k, PrvKey, Tree): and *ciphertexts* with *document indexes*. It guarantees that no adversary can distinguish a document index from another one. That is, the document index does not reveal any information about the underlying keywords to any adversary. The security game for IND-SPKE-CKA is defined as,

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

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

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

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

$$docIndex_b \leftarrow genDocIndex(f_b, PubKey) \tag{1}$$

C sends $docIndex_b$ 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 index of f_0 or f_1, $\mathcal{A}$ wins the game. The advantage that $\mathcal{A}$ wins the game is defined as $Adv_{\mathcal{A}}^{IND\text{-}SPKE\text{-}CKA} = Pr[b' = b] - \frac{1}{2}$.*

3.2.2 Security of Search Keyword. We introduce a security game for the search keyword's security, namely indistinguishability under the chosen search keyword attack for SPKE (IND-SPKE-CSKA). It guarantees that the adversary can not identify search keywords even if they are repeated. We define the security game for IND-SPKE-CSKA as,

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

Setup. *C runs the $KeyGen(\mathcal{P})$ 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 docIndex of any document f_i or initiate and retrieve search results for any keyword w of it's choice from C.*

Challenge. *$\mathcal{A}$ sends C two keywords w_0 and w_1 of its choice which it wishes to be challenged and which have not been searched before. C picks $b \leftarrow \{0,1\}$. C initiates search for w_b and send the search result (document index) $docIndex_b$ 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 $Adv_{\mathcal{A}}^{IND-SPKE-CSKA} = Pr[b' = b] - \frac{1}{2}$.*

3.2.3 Forward Privacy. Forward privacy ensures the previous search leakages cannot be exploited to infer information about the newly added documents [28]. In our scheme, the update operation is defined as deletion of the old document from the cloud and adding the updated document (details in 5.2.4). Therefore, we will only discuss the Forward Privacy of adding a document. Formally, we define the forward privacy game FS-SPKE of our scheme as,

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

Setup. *C runs the $KeyGen(\mathcal{P})$ 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 it's choice including w_0 and w_1 from C.*

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

$$docIndex_b \leftarrow genDocIndex(f_b, PubKey) \tag{2}$$

C sends $docIndex_b$ 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 $Adv_{\mathcal{A}}^{FS-SPKE} = Pr[b' = b] - \frac{1}{2}$.*

4 PRELIMINARIES

4.1 Encryption Technique

Homomorphic encryption (HE) 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 [1, 25]. **Fully Homomorphic Encryption (FHE)** is still not practical to use in today's big data world as it is impractically slow. It uses complex and computationally heavy operations. Therefore, we propose to use **Partially Homomorphic Encryption (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 key leakage. Therefore symmetric key encryption is not suitable.

Additive and Subtractive Homomorphism: The encryption supports an addition operator $\boxplus$ and a subtraction operator $\boxminus$ such that for $Enc(a, PubKey) \rightarrow e_a$ and $Enc(b, PubKey) \rightarrow e_b$, $Dec(e_a \boxplus e_b, PrvKey) \rightarrow a + b$ and $Dec(e_a \boxminus e_b, PrvKey) \rightarrow a - b$. Subtraction is used for optimization, however this is an optional feature.

Indistinguishability under Chosen Ciphertext Attack (IND-CCA1): We assume the 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 [2, 21]. Although, an Indistinguishability under Chosen Plaintext Attack (IND-CPA) secure PHE can also work in RS-PKE with fewer security guarantees by modifying Definition 1 and 2. The security game of IND-CCA1 is given in Appendix A.

4.2 Notations

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

4.3 Keyword Balanced Binary Tree (KBB tree) Construction

KBB tree [33] is a dynamic balanced binary tree where each node Node can be defined as:

$$\texttt{Node} \leftarrow \langle\ \texttt{ID, Data, Left, Right, Parent}\ \rangle$$

where,
ID denotes unique identifier for each tree node,
Data is a vector with length N (size of the keyword dictionary),
Left, Right and Parent indicate the left, right, and parent node.

In the leaf level, node LeafNode denotes a specific document f and LeafNode.Data[i] stores the normalized TF score of keyword w_i in f. For internal node InternalNode,
InternalNode.Data[i] ← max(InternalNode.Left.Data[i], InternalNode.Right.Data[i]), i= 1, ..., N

The owner can perform multi keyword ranked search in Xia et al.'s scheme. The owner first creates a vector *trapdoor* for the searched keywords and sends it to the cloud. The relevance score *RScore* of any node for that query in the index tree is calculated using $RScore \leftarrow Data \cdot trapdoor$. The cloud then performs a recursive procedure upon the tree, named as "Greedy Depth-first Search" (GDFS). The cloud uses *RScore* to greedily select which nodes are accessed during GDFS and return the top-k documents with the largest relevance scores for that query to the owner.

However, with this approach, the cloud can store the *trapdoor* and use it extensively without the owner's knowledge. The relevance scores are leaked to the cloud, which might be used in leakage abuse attacks. If the document collection in owner side updated, the KBB tree in the cloud also need to be updated. In order to reduce the communication overhead, the data owner stores a copy of unencrypted index tree. The owner modifies the KBB tree in his end and send updates it to the cloud.

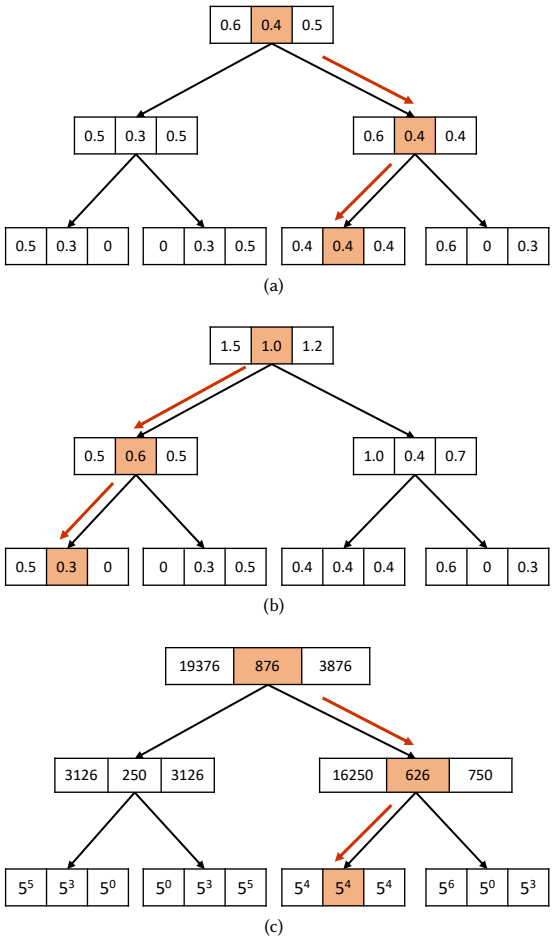

Figure 2: An example of the KBB tree-based indexes with 4 documents, the cardinality of the dictionary is 3, and search is for the topmost relevant document for the second word. In all cases, the search starts at the root node and reaches the leaf node by following the node's path with the maximum value. Figure (a) is the original KBB tree, where leaf nodes represent documents and the internal nodes are generated by taking the *maximum* value between the child nodes. This tree is generated at owner side on plaintext data. Figure (b) is the same tree; however, we used *addition* operation to generate the internal nodes instead of *maximum* operation. As our proposed tree is generated dynamically on cloud and we need to operate on encrypted data, we cannot do *maximum* operation. However, *addition* cannot preserve the relative ordering between nodes required for searching. In figure (c), the data values of the leaf nodes are applied to function $b()$ with $\alpha = 10$ and $M = 5$, and then the internal nodes are constructed like figure (b). Theorem 1 enables us to use the same searching mechanism as the original KBB tree by preserving relative ordering through homomorphic addition.

5 RS-PKE CONSTRUCTION

In this paper, we develop a secure SPKE 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 admin. Each entity plays a different role in the data outsourcing environment. The document producer encrypts the document and creates an encrypted document index. Both of them are outsourced to the cloud. The cloud then generates an encrypted dynamic KBB tree-based index from the encrypted document indexes according to the techniques discussed in 5.1. 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. An admin can initiate a search request for a specific keyword and retrieve the best matched (highest TF score) documents through an interactive search protocol.

5.1 Proposed Modified KBB Tree

The KBB tree of RS-PKE is based on Xia et al.'s KBB tree. Our node and tree structures are similar to Xia et al. However, our design goals are different. For supporting lightweight platforms like IoT: (1) The value of the nodes should be encrypted using the public key and decrypted using the private key, (2) The lightweight document producers should not store the unencrypted index tree as the owner does in Xia et al, and (3) The encrypted document index generation for a specific document in the document producers' side should be faster and should not require any information about the existing

tree like Xia et al. The document producer should be able to generate encrypted documents and document index, upload, and delete them without keeping any information. Xia et al. focused on faster initial tree generation and search, where we focused on a faster dynamic update of documents. In addition, we wanted to make RS-PKE forward secure and not leak relevance scores to the cloud.

We found out that partial homomorphic encryption can be used to encrypt document indexes (normalized TF values), and the cloud can operate on encrypted document indexes to generate the KBB tree. This encrypted document index is stored in `Data` vector of the leaf nodes. The internal nodes are generated by the homomorphic addition of the left and the right child nodes.
$InternalNode.Data[i] \leftarrow$
$InternalNode.Left.Data[i] \boxplus InternalNode.Right.Data[i]$,
$i = 1, \ldots, N$

We cannot take *maximum* of left and right child nodes to generate the internal node like the original KBB tree as we are operating on partially homomorphically encrypted data in the cloud. Doing *maximum* operation on encrypted data would require the document indexes to be encrypted using FHE which is vey slow. The *maximum* operation was possible in Xia et al.'s scheme as the KBB tree was generated on the owner side on plaintext data.

However, this internal node construction process nullifies relative order information between nodes which is required during searching (example in Fig. 2(b)). To preserve the relative ordering thorough addition, 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. For this, we exploit the fact that, if $max(a, b) > max(c, d)$ then for a sufficiently large M,

$$M^a + M^b > M^{max(a,b)} \geq M^{max(c,d)+1} = M.M^{max(c,d)} > 2.M^{max(c,d)} > M^c + M^d$$

Using this, we define a function $b(): \mathbb{N}_0 \rightarrow \mathbb{N}_0$ to preserve relative order through the sum.

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

$$b(s) = M^s \tag{3}$$

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 function $b()$ 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.*

The proof of Theorem 1 is given in Appendix B. The transformed relevance scores are encrypted to create document indexes (details in 5.2.3) and the KBB tree is constructed in cloud using partial homomorphic addition (details in 5.2.4). The two properties of $b()$ ensure that the search procedures described in 5.2.5 give correct results (example in Fig. 2(c)).

5.2 System Description

5.2.1 ***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.

5.2.2 ***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 the admin, respectively.

5.2.3 ***Encrypted Document and Document Index 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 index. For measuring the relevance of a document to a keyword, we are using *Term Frequency (TF)*, which indicates the weight of a keyword in a document. Formally, the encrypted document index of a document f, is constructed using the following equation:

$$\begin{aligned} E = &Enc(b(\lfloor \alpha * TF_1 \rceil), PubKey), Enc(b(\lfloor \alpha * TF_2 \rceil), PubKey), \\ &Enc(b(\lfloor \alpha * TF_3 \rceil), PubKey), \ldots, Enc(b(\lfloor \alpha * TF_N \rceil), PubKey) \end{aligned} \tag{4}$$

Here, $TF_i, i \in [1 - N]$ denotes TF score of i^{th} word in the dictionary for f. TF_i, is defined as: $TF_i = \frac{FR_i}{\max_k FR_k}$. Here, FR_i is the frequency of each keyword $w_i \in W$ and $\max_k FR_k$ is the maximum frequency of any keyword $w_k \in W$ in document f. The TF score is then normalized by multiplying with α and rounded to an integer in $[0 - \alpha]$ range to be applied to function $b()$ of equation 3. The output of function $b()$ is then encrypt using the $PubKey$.

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

5.2.4 ***Dynamic KBB Tree Construction***. In Section 5.1, 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:

`Node ← ⟨ ID, Data, Left, Right, Parent ⟩`

where, `ID` denotes unique identifier for each tree node, `Data` is the encrypted document index for leaf nodes, the homomorphic summation of it's left and right child's data for other nodes. `Left`, `Right` and `Parent` indicates the left, right, and parent node, respectively.

Insertion: A leaf node in the index tree is introduced for each document in the collection with document `ID` and encrypted document index as the `Data`. The internal tree nodes are computed by adding it's left child and right child's data values using the addition of PHE. The formal construction process of the index tree is presented in Algorithm 2. In Algorithm 2, a new document index 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 admin 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.

5.2.5 ***K Ranked Documents Search***. K ranked documents retrieval for a specific keyword involves both the cloud and the admin. The admin can initiate search process for the i^{th} keyword w_i to retrieve top-k documents (highest TF values for the searched keyword). In this search process, the searched keyword is kept to the admin. The search process is a recursive depth-first search (DFS) procedure upon the tree starting from the root node. However, as the search keyword is kept in admin side, admin selects which nodes to traverse in this DFS. Cloud only provides access to specific nodes like in Algorithm 4.

The formal ranked search procedure from the admin side is presented in Algorithm 5. We construct a result queue denoted as $priorityQueue$, whose element is defined as $(score, id)$. The $priority-Queue$ stores the id of the k accessed documents with the largest relevance scores $score$ to the query. The elements of the $priority-Queue$ are ranked in descending order according to the $score$. It is updated during the search procedure.

The admin starts the ranked search procedure taking the root node's value from the cloud and calculating the $minScore$, the minimum relevance score for top-k ranked documents. 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. From the tree construction process, 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 compute $minScore$ 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 $M^{number\ of\ digits\ left}$. From Fig. 3 we can see that, at first the admin requests the value of the root node from the cloud. After that, the admin converts the root node's 1^{st} value (the search is for 1^{st} keyword) 1201 into a M (10) base number. 1201 can be written uniquely as 1000*1+ 100*2+10*0+1*1. We can see from 1000*1+ 100*2+10*0+1*1 that the tree contains one node containing value 1000, two nodes containing value 100, and one node of value

Algorithm 1: genDocIndex

Data: Document f
Result: Encrypted f with encrypted document index E to the cloud
foreach $w \in W$ **do**
  compute tf for w ;
  $h \leftarrow b(\lfloor \alpha * tf \rceil), e \leftarrow Enc(h, PubKey)$;
  Insert e to E ;
end
Send (Encrypted f, E) to cloud;

Algorithm 2: buildIndexTree

Data: encryprted document c, doc index $data$
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 it's $parent$;
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 $freeNodes$;

Algorithm 4: searchCloud

Data: id of the node to be traversed from admin
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

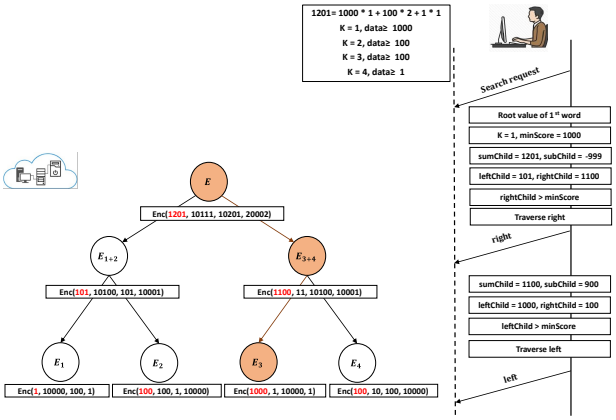

Figure 3: An example of the search process that returns the most relevant document for the 1^{st} keyword.

1. Hence, if k=1, data values less than 1000 need not be traversed, if k=2, values less than 100 need not be traversed and so on.

Then, the admin calculates the next node to be traversed after the root node. The admin can calculate the values of the child nodes without accessing those nodes. In RS-PKE, the internal nodes are the summation of their child nodes and the admin can obtain the subtraction of the child nodes from the cloud. The summation 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 whole tree is covered.

Algorithm 5: rankedsearchAdmin

Data: Searching for k documents for the i^{th} word $w_i \in \boldsymbol{W}$, 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`$(id, sumChild)$**:**
  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 $(id, sumChild)$ to $priorityQueue$;
    end
    else
      if $sumChild$ *is greater than smallest element of* $priorityQueue$ **then**
        Delete the element with smallest relevance score from $priorityQueue$;
        Insert $(id, sumChild)$ to $priorityQueue$;
      end
    end
  end
  else
    $subChild \leftarrow decrypt(difference[i])$;
    $leftChild \leftarrow \frac{sumChild+subChild}{2}$,
    $rightChild \leftarrow \frac{sumChild-subChild}{2}$;
    if $leftChild \geq minScore$ **then**
      $leftId \leftarrow$ current node's left child's id from cloud;
      $dfsRanked(leftId, leftChild)$;
    end
    if $rightChild \geq minScore$ **then**
      $rightId \leftarrow$ current node's right child's id from cloud;
      $dfsRanked(rightId, rightChild)$;
    end
  end

Fig. 3 shows an example of ranked search where admin searches for $k = 1$ document for the 1^{st} keyword. The admin first computes $minScore = 1000$. For root node E, $sumChild = 1201$ is the summation of its child nodes and the admin gets the subtraction $subChild = -999$ from the cloud. Then admin calculates left child's value $(sumChild + subChild)/2 = 101$ and the right child's value $(sumChild - subChild)/2 = 1100$. As only $E_{(3+4)}$ has values greater than 1000, therefore, only the left node is traversed after root node. This process goes on and finally leaf node E_3 is the search result. The admin takes the document that corresponds to E_3 from the cloud.

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

5.3 Leakage

RS-PKE reveals some information to the cloud server for efficiency, like all other practical SE schemes. The leakage of our scheme $\mathcal{L}_{RS-PKE}$ 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 RS-PKE, 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 index 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 index). Therefore, $\mathcal{L}_{BuildIndexTree} = (N, |c|, id)$.

$\mathcal{L}_{Search}$. During the interactive `Search` procedure between the cloud and the admin, 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 [17]. The leakage of access pattern in RS-PKE implies that for same document collection, for same keyword, same ids are returned as a search result. Therefore, $\mathcal{L}_{Search} = (id_1, id_2, id_3, \ldots, id_k)$.

5.4 Security Analysis

In RS-PKE, 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 3.2 against an "honest but curious" cloud server.

IND-SPKE-CKA: The formal security claim of the IND-SPKE-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-SPKE-CKA** security of our scheme, there is a PPT adversary $\mathcal{B}$ who can break the **IND-CCA1** security of underlying PHE.*

IND-SPKE-CKA security of our scheme depends on the IND-CCA1 security of the underlying PHE scheme. The proof is postponed to Appendix C.1.

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

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

IND-SPKE-CSKA depends on the fact that $\mathcal{L}_{Search}$ is only limited to access pattern, which makes IND-SPKE-CSKA a restricted version of IND-SPKE-CKA. The proof is postponed to Appendix C.2.

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

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

IND-SPKE-CKA security of RS-PKE is sufficient to ensure forward privacy. The proof is given to Appendix C.3.

6 PERFORMANCE EVALUATION

We have implemented the RS-PKE using Java language. Our entities (document producer, cloud server, and admin) are Java applications (jar) that connect through the REST protocol using Spring Framework. We have used Paillier PHE scheme [23], to encrypt and decrypt the document indexes generated from document producers. Javallier[1], 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 [16]. It is a widely used dataset in SE literature [24, 26, 31–33]. 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 4 core processor with 32GiB memory) instance as the *cloud server* and *t2.micro* (Intel Xeon 3.3 GHz 1 core processor with 1GiB ram) instance for *admin*. And we took the lowest possible AWS instance *t2.nano* (Intel Xeon 3.3 GHz 1 core processor with 0.5GiB ram) instance for *document producer*. The test includes: (1) the efficiency and storage requirements of single encrypted document index generation, (2) the efficiency and storage requirements of

[1] https://github.com/n1analytics/javallier

dynamic index construction, and (3) the efficiency and accuracy of the search. We make our implementation open-source[2].

6.1 Encrypted Document Index Generation

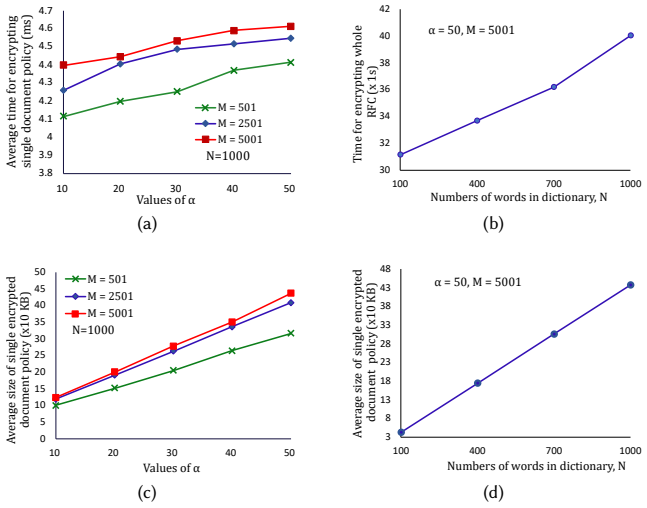

Figure 4: Average time cost for generating encrypted single document index (a) & (b) and average size cost for encrypted single document index (c) & (d).

The encrypted document index 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 PHE*: 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 index. Hence, the size complexity of encrypted document index is $O(N\alpha \log M)$. The time cost for encrypting a single document index is proportional to the single encrypted document index's size. Fig. 4 shows how the average time for creating an encrypted document index and the average size of encrypted document index changes with different parameters.

6.2 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 node is accessed at a level. Consequently, time changes logarithmically with the insertion of documents.

[2]https://github.com/pkse-searcher/secure-pkse

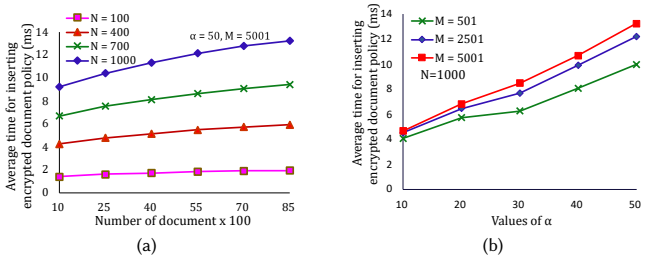

Figure 5: The average time cost for inserting a node in the tree. For (b) the collection of document is fixed, $|F|$= 8582.

(2) *Size of an encrypted document index*: When a leaf node is inserted, a partial homomorphic addition is performed for each element in the document index 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 index 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. 5 shows the insertion time. The graphs of deletion are moved to Appendix 10.

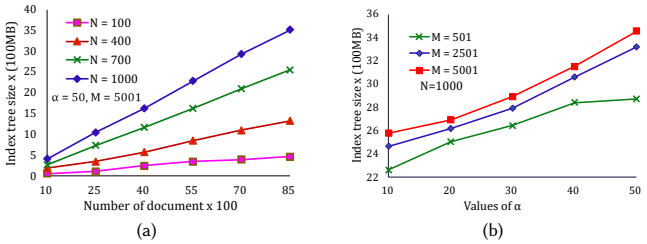

Figure 6: (a) & (b) is the size of the KBB tree. For (b) the collection of document is fixed, F = 8582.

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 index, and the document index size is proportional to $N\alpha log(M)$. Therefore, the storage requirement of index tree is $O(|F|N\alpha \log M)$. Fig. 6 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.

6.3 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.

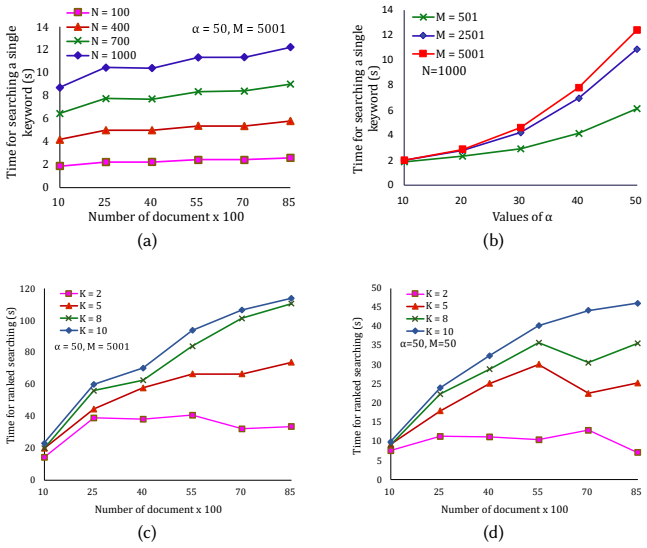

Figure 7: Average time cost for searching: single document (a) & (b) and k ranked documents (c) & (d). For (b) the collection of document is fixed, F = 8582.

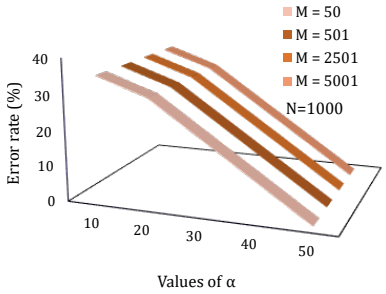

Figure 8: A 3D graph representing the error rate in search results for different α and M with fixed documents and dictionary, N = 1000.

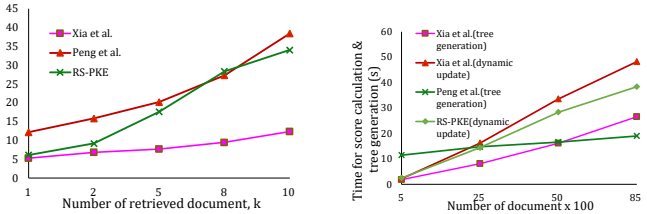

Figure 9: Average time for (a) single keyword search for different k and (b) score calculation & tree generation for different F in different ranked search schemes.

(2) *Network load*: The network load is proportional to the size of an encrypted document index, 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. 7 shows the time cost of searching that returns a single document and k ranked documents with different parameters, respectively.

6.4 Search Accuracy

The search precision is defined by the rate of deviation in fetching correct documents for a specific keyword. 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. 8 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. 7 (c) and (d), we can see that decreasing M to 10% decrease the search time to 25-30%.

6.5 Performance Comparison

We compare the performance of RS-PKE with Xia et al.[33], and Peng et al.[24]. We are comparing our scheme with two SSE schemes instead of other IoT supported SPKE because no other IoT supported SPKE supports top-k ranked search. Xia et al.'s scheme is not efficient for IoT as the number of tree nodes are fixed. The tree is generated initially with all the documents. For dynamic new document insertion, the tree needs to delete an old document and then insert the new one in its place. In Fig. 9(b), we show comparison of both initially generating tree with all documents and regenerating the tree with dynamic updates. The dynamic update graph of Xia et al. is more comparable with RS-PKE as all our tree operations are dynamic. Xia et al. also requires the owner to perform on the unencrypted index tree to generate the update. Storing the index tree in IoT's end and updating it is not feasible for IoT devices. From Fig. 9, we can see that Xia et al.'s [33] scheme performs better in search, whereas RS-PKE is better at generating the tree with dynamic updates. RS-PKE also does not need the document producers to store any details. Peng et al.'s [24] 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. However, they do not support dynamic updates from the same user. SSE schemes are fast due to their faster symmetric-key encryption; however, they are not suitable for lightweight platforms for the risk of key leakage. The mentioned schemes also do not support forward security. Our goal is to make an SPKE for IoT, which will support top-k ranked search with forwarding security without being impractically slow. We think RS-PKE is on the right path for implementing SE in the cloud on encrypted data from lightweight platforms.

7 CONCLUSIONS

This paper proposes a secure, forward private, efficient, and dynamic SPKE search framework over encrypted cloud data that supports single-keyword ranked search. We analyze the security

of encrypted document indexes, 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, RS-PKE'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 [20] in RS-PKE 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.

REFERENCES

[1] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. 2018. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 79.

[2] Frederik Armknecht, Stefan Katzenbeisser, and Andreas Peter. 2013. Group homomorphic encryption: characterizations, impossibility results, and applications. *Designs, codes and cryptography* 67, 2 (2013), 209–232.

[3] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2020. Revisiting Leakage Abuse Attacks.. In *Proceedings 2020 Network and Distributed System Security Symposium*.

[4] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. 2004. Public Key Encryption with Keyword Search. In *Advances in Cryptology - EUROCRYPT 2004*, Christian Cachin and Jan L. Camenisch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 506–522.

[5] Raphael Bost. 2016. $\sum o\varphi o\varsigma$: Forward Secure Searchable Encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 1143–1154. https://doi.org/10.1145/2976749.2978303

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

[7] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. 2013. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on parallel and distributed systems* 25, 1 (2013), 222–233.

[8] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 668–679.

[9] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In *CCS 2018 : The 25th ACM Conference on Computer and Communications Security*. 1038–1055.

[10] Biwen Chen, Libing Wu, Neeraj Kumar, Kim-Kwang Raymond Choo, and Debiao He. 2019. Lightweight Searchable Public-key Encryption with Forward Privacy over IIoT Outsourced Data. *IEEE Transactions on Emerging Topics in Computing* (2019).

[11] Biwen Chen, Libing Wu, Huaqun Wang, Lu Zhou, and Debiao He. 2020. 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 (2020), 5813–5825.

[12] Rongmao Chen, Yi Mu, Guomin Yang, Fuchun Guo, and Xiaofen Wang. 2015. A new general framework for secure public key encryption with keyword search. In *Australasian Conference on Information Security and Privacy*. Springer, 59–76.

[13] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '06)*. Association for Computing Machinery, New York, NY, USA, 79–88. https://doi.org/10.1145/1180405.1180417

[14] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. 2018. Efficient dynamic searchable encryption with forward privacy. *Proceedings on Privacy Enhancing Technologies* 2018, 1 (2018), 5–20.

[15] Florian Hahn and Florian Kerschbaum. 2014. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 310–320.

[16] I.E.T.F. 2020. Request for Comments (RFC). https://www.rfc-editor.org/retrieve/bulk/. [Online; accessed 02-February-2020].

[17] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS*, Vol. 20. Citeseer, The Internet Society, Virginia, United States, 12.

[18] Tasnim Kabir and Muhammad Abdullah Adnan. 2017. 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)*. IEEE, ICPS, Dhaka, Bangladesh, 1–9.

[19] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. 2017. Forward secure dynamic searchable symmetric encryption with efficient updates. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, ACM, New York, NY, USA, 1449–1463.

[20] Ágnes Kiss, Masoud Naderpour, Jian Liu, N Asokan, and Thomas Schneider. 2019. SoK: Modular and efficient private decision tree evaluation. *Proceedings on Privacy Enhancing Technologies* 2019, 2 (2019), 187–208.

[21] Helger Lipmaa. 2010. On the cca1-security of elgamal and damgård's elgamal. In *International Conference on Information Security and Cryptology*. Springer, 18–35.

[22] Yinbin Miao, Jianfeng Ma, Ximeng Liu, Jian Weng, Hongwei Li, and Hui Li. 2019. Lightweight Fine-Grained Search Over Encrypted Data in Fog Computing. *IEEE Transactions on Services Computing* 12, 5 (2019), 772–785.

[23] Pascal Paillier. 1999. Public-key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques* (Prague, Czech Republic) *(EUROCRYPT'99)*. Springer-Verlag, Berlin, Heidelberg, 223–238. http://dl.acm.org/citation.cfm?id=1756123.1756146

[24] Tianyue Peng, Yaping Lin, Xin Yao, and Wei Zhang. 2018. An efficient ranked multi-keyword search for multiple data owners over encrypted cloud data. *IEEE Access* 6 (2018), 21924–21933.

[25] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. 1978. On data banks and privacy homomorphisms. *Foundations of secure computation* 4, 11 (1978), 169–180.

[26] Mukalel Bhaskaran Smithamol and Rajeswari Sridhar. 2018. PECS: Privacy enhanced conjunctive search over encrypted data in the cloud supporting parallel search. *Computer Communications* 126 (2018), 50–63.

[27] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (SP '00)*. IEEE Computer Society, USA, 44.

[28] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage.. In *NDSS*, Vol. 71. The Internet Society, Virginia, United States, 72–75.

[29] Shi-Feng Sun, Xingliang Yuan, Joseph K Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. 2018. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 763–780.

[30] Wenhai Sun, Bing Wang, Ning Cao, Ming Li, Wenjing Lou, Y Thomas Hou, and Hui Li. 2013. Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. *IEEE Transactions on Parallel and Distributed Systems* 25, 11 (2013), 3025–3035.

[31] Cong Wang, Ning Cao, Jin Li, Kui Ren, and Wenjing Lou. 2010. Secure Ranked Keyword Search over Encrypted Cloud Data. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS '10)*. IEEE Computer Society, USA, 253–262. https://doi.org/10.1109/ICDCS.2010.34

[32] Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. 2011. Enabling secure and efficient ranked keyword search over outsourced cloud data. *IEEE Transactions on parallel and distributed systems* 23, 8 (2011), 1467–1479.

[33] Zhihua Xia, Xinhui Wang, Xingming Sun, and Qian Wang. 2015. A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE transactions on parallel and distributed systems* 27, 2 (2015), 340–352.

[34] Xiaojun Zhang, Chunxiang Xu, Huaxiong Wang, Yuan Zhang, and Shixiong Wang. 2019. 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).

[35] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 707–720. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang

[36] Yuan Zhang, Chunxiang Xu, Jianbing Ni, Hongwei Li, and Xuemin Sherman Shen. 2019. Blockchain-assisted public-key encryption with keyword search against keyword guessing attacks for cloud storage. *IEEE Transactions on Cloud Computing* (2019).

A INDISTINGUISHABILITY UNDER ADAPTIVE CHOSEN CIPHERTEXT ATTACK

The *IND* notation provides security definitions in terms of games. Here a challenger keeps his key secret but gives some specific capabilities to any adversary, and an adversary, with the help of that capabilities, tries to break the security of the encryption.

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.*

The advantage of a adversary in IND-CCA1 is, $Adv^{IND-CCA1}_{adversary} = Pr[guess = b] - \frac{1}{2}$. Any scheme is IND-CCA1 secure if no adversary has a non-negligible advantage in winning the above game.

B PROOF OF THEOREM 1

Proof. For the first property,

$$b(0) = M^0 = 1 \tag{5}$$

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,

$$b(s) = M * M^{s-1} = M * b(s-1) > (M-1) * b(s-1) \tag{6}$$

Hence, $b(s) > (M-1) * b(s-1)$ ensures the second property $b(i_{max}) > \sum b(j)$ 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. □

C SECURITY PROOFS

C.1 Proof of Theorem 2

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

Setup: 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 index generation/encryption. $\mathcal{A}$ sends document f to $\mathcal{B}$. $\mathcal{B}$ generates the *plaintext document index* vector, encrypts each element from C, combines them to $docIndex$ 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 C for decryption.

Challenge: $\mathcal{A}$ sends two documents f_0 and f_1 to $\mathcal{B}$. $\mathcal{B}$ generates two *plaintext document index* 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 C. C picks a random value of $b \leftarrow \{0,1\}$ and use the same b throughout the challenge. For all positions, C sends back $Enc(v_{b,i})$ to $\mathcal{B}$. $\mathcal{B}$ combines them to a $docIndex$ 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 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-SPKE-CKA game in our scheme. If $\mathcal{A}$ can break IND-SPKE-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 $Adv^{IND-SPKE-CKA}_{A}$. Hence, the IND-SPKE-CKA of our scheme depends solely on the security of underlying PHE.

C.2 Proof of Theorem 3

As shown in Section 5.3, the search leakage $\mathcal{L}_{Search}$ of our scheme is only limited to access pattern. Hahn and Kerschbaum [15] and later others [19, 29] 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 index, as discussed below.

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

C.3 Proof of Theorem 4

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

D DELETION IN TREE

The performance analysis of the deletion algorithm is given in Fig. 10.

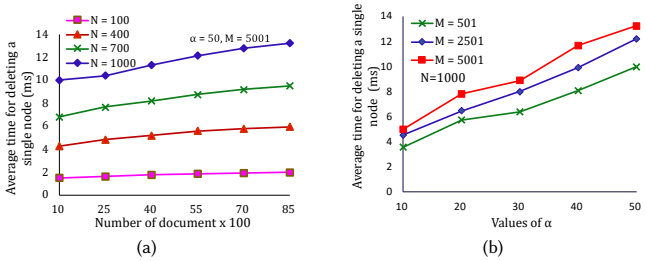

Figure 10: Average time cost for deleting a node. For (b), F = 8582 **is fixed.**