2015 IEEE 35th International Conference on Distributed Computing Systems

Privacy Preserving String Matching for Cloud Computing

Bruhadeshwar Bezawada*, Alex X. Liu*†, Bargav Jayaraman‡, Ann L. Wang† and Rui Li*§
*National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
†Dept. of Computer Science and Engineering, Michigan State University, East Lansing, MI 48864, USA
‡International Institute of Information Technology, Hyderabad, India
§College of Information Science and Engineering, Hunan University, China
Email: bru@nju.edu.cn, (alexliu, liyanwan)@cse.msu.edu, bargav.jayaraman@gmail.com, lirui@hnu.edu.cn

***Abstract*—Cloud computing has become indispensable in providing highly reliable data services to users. But, there are major concerns about the privacy of the data stored on cloud servers. While encryption of data provides sufficient protection, it is challenging to support rich querying functionality, such as *string matching*, over the encrypted data. In this work, we present the first ever symmetric key based approach to support privacy preserving string matching in cloud computing. We describe an efficient and accurate indexing structure, the PASStree, which can execute a string pattern query in logarithmic time complexity over a set of data items. The PASStree provides strong privacy guarantees against attacks from a semi-honest adversary. We have comprehensively evaluated our scheme over large real-life data, such as Wikipedia and Enron documents, containing up to 100000 keywords, and show that our algorithms achieve pattern search in less than a few milliseconds with 100% accuracy. Furthermore, we also describe a relevance ranking algorithm to return the most relevant documents to the user based on the pattern query. Our ranking algorithm achieves 90%+ above precision in ranking the returned documents.**

***Keywords*-Data Privacy, String Matching, Prefix Matching, Secure Index, Cloud Storage, IND-CKA, SSE**

I. Introduction

A. Motivation

Advances in cloud computing have redefined the landscape of modern information technology as seen by the popularity of cloud service providers such as Amazon, Microsoft Azure and Google App Engine. Data storage outsourcing is a prime application in cloud computing, which allows organizations to provide reliable data services to users without concerning with the data management overheads. But, there are several practical apprehensions regarding the privacy of the outsourced data and these concerns are a major hindrance towards the uninhibited adoption of cloud storage services. Generally, the cloud service provider does not attempt to violate data privacy, there are internal factors like malicious employees who will abuse the client data at the first available opportunity. Also, cloud computing servers are prone to serious external threats like malware and botnets, which may not be reported publicly.

In this paper, we consider the popular cloud storage model, which is composed of three entities: the data owner, the cloud server and the authorized users. The data owner stores the data on the cloud server and authorizes the users to issue different queries on the outsourced data. To protect the data, the data owner encrypts the data prior to outsourcing and shares the decryption keys with the authorized users. However, the encryption is a major hindrance to perform regular data access operations, such as searching for documents containing specific keywords or patterns. For instance, the Google$^{(tm)}$ search engine supports rich functionality like, find documents containing, *"terms appearing in the text of the page"*, *"terms appearing anywhere in title of the page"* and so on. Therefore, there is a crucial need for techniques to support a rich set of querying functionality on the encrypted data without violating the privacy of the users.

B. Problem Statement

Our work focuses on the problem of privacy preserving *string pattern* queries on keywords in outsourced documents or database records. A *string pattern* query is a sequence of characters. A keyword is said to *match* a string pattern query, if the query string is either identical to the keyword or is *contained* as a sub-string of the keyword. For example, given a string pattern query, "*late*" where the "*" denotes any other characters, a sample list of matching keywords are: *"ablate, contemplate, plates, elated"* and so on. In other words, it is necessary to examine every possible sub-string [1], [2] of the keyword as the query string can occur anywhere inside the keyword.

To describe the problem in the context of cloud computing, initially, the data owner has a set of documents, $\mathbf{D} = \{D_1, \cdots, D_n\}$ where each document contains a set of distinct keywords, $D_i(W) = \{w_1, \cdots, w_p\}$. Before outsourcing the data to the cloud, the data owner computes an index $\mathbf{I}$ on the documents' keywords, encrypts the documents and stores the encrypted documents, along with the index, on the cloud server. Now, to search for query string pattern in these encrypted documents, an authorized user computes an encrypted query or a "*trapdoor*", using the string pattern query p, and submits it to the cloud

This work is supported in part by the National Science Foundation under Grant Number CNS-1318563, CCF-1347953, and CNS-1017598, the National Natural Science Foundation of China under Grant Numbers 61472184, 61321491, 61370226, and 61272546, the Jiangsu Future Internet Program under Grant Number BY2013095-4-08, the Jiangsu High-level Innovation and Entrepreneurship (Shuangchuang) Program, and the China Postdoctoral Science Foundation.

server. The server processes the trapdoor on the index $\mathbf{I}$ and retrieves all documents, D_i such that, p matches at least one keyword $w \in D_i(W)$ for $1 \leq i \leq n$. There are two major challenges in this process: (a) the index $\mathbf{I}$ should not leak any information regarding the keywords (*e.g.*, size and content) and (b) the string query processing technique should be efficient and scalable to large collections of keywords.

C. Adversary and Security Model

We adopt the semi-honest adversary model [3] for the cloud server and any passive adversaries. In this model, the cloud server *honestly* adheres to the communication protocols and the query processing algorithms, but is *curious* to learn additional information about the user by observing the data processed in the search protocol. Our goal is to construct a *secure index*, which is secure in the strong *semantic security against adaptive chosen keyword attack* (IND-CKA) security model described in [4]. We assume that the index and the trapdoor construction relies on symmetric key encryption algorithms, *i.e.*, our solution is a symmetric searchable encryption (SSE) [5], [6] scheme. To prove security in IND-CKA model, an adversary is given two distinct document sets $\mathbf{D_0}$ and $\mathbf{D_1}$, containing nearly equal number of keywords and with some amount of overlap and *one* secure index $\mathbf{I_b}$ for $\mathbf{D_b}$ where $b = 0$ or $b = 1$. Finally, the adversary is challenged to output the correct value of b with non-negligible probability. An IND-CKA secure index does not necessarily hide the number of keywords in a document, the *search* patterns of the users, *i.e.*, the history of the trapdoors used, and the *access* patterns, *i.e.*, the documents retrieved due to the search queries.

D. Limitation of Prior Art

Prior secure string pattern matching schemes under the secure multi-party computation model have high communication and computational complexity (*e.g.*, exponentiations and garbled circuit evaluations) [7]–[10].

Several symmetric key based searchable encryption techniques [4]–[6], [11]–[16] have focused on privacy preserving *exact* keyword matching in cloud computing. We note that, keyword matching problem is an instance of the string pattern matching problem where the pattern query is a *whole* keyword, *i.e.*, the complete query must match a stored keyword. The work in [17] proposed a similarity based keyword matching solution, in which the cloud server retrieves keywords *similar* to a given query string by using a predefined hamming distance as the similarity measure. However, the similarity of string pattern query cannot be measured by hamming distance as a matching keyword can be of arbitrary length and we are only interested in the *exact* matching of a query sub-string within the keyword.

All the SSE solutions discussed above cannot address the privacy preserving string query pattern matching problem considered in our work. In this work, for the first time ever, we describe an efficient privacy preserving approach for executing string pattern queries over encrypted cloud data. Unless specified otherwise, we will use the term "string pattern" and "pattern", interchangeably in our work.

E. Proposed Approach

Our approach has two building blocks: a string pattern matching mechanism with strong privacy guarantees and an efficient index structure for fast processing of string pattern queries. First, our pattern matching approach is based on a simple intuition, *i.e.*, if a pattern query matches a keyword then the pattern *must* be a sub-string of the keyword. Therefore, for each keyword in the document data set, we encrypt *every* possible sub-string of the keyword and store the encrypted sub-strings in the secure index. With this approach, we reduce the problem of pattern matching to that of comparing a query trapdoor with the encrypted sub-strings of the keyword. Second, to achieve fast searching, we arrange the encrypted sub-strings in a highly balanced binary tree structure, the *Pattern Aware Secure Search* tree (PASStree), in which each tree node corresponds to a Bloom filter encoding of a set of the encrypted sub-strings. The PASStree is built in a top-down manner. The root node of the PASStree stores all the encrypted sub-strings corresponding to all the keywords. Any two child nodes store an equal sized partition of the set of sub-strings stored in their parent node and each leaf node corresponds to the encrypted sub-strings of a distinct keyword. This ensures that the PASStree satisfies the index indistinguishability [4] property as the size of each Bloom filter depends only on the number of encrypted sub-strings stored and not on the content of the sub-strings.

F. Technical Challenges and Solutions

The first major challenge is, improving the efficiency of search on the PASStree by imposing some order on the encrypted sub-strings. Without any ordering, every pattern query can induce a worst case search complexity scenario over the PASStree causing a major performance bottleneck. To overcome this, we devise a novel similarity based clustering technique to cluster the keywords in the PASStree and narrow the search towards the matching keywords. The similarity between two keywords is calculated using multiple independent dimensions to prevent any leakage of content similarity due to the PASStree organization.

The second major challenge is, to retrieve the most relevant documents for a given pattern query, because a pattern query can match any sub-string at any location in a keyword and hence, it is essential to *rank* the matching keywords by relevance. Towards this, we leverage the structure of the PASStree in the following manner: for each regular Bloom filter in the PASStree, we associate an additional Bloom filter, called the *sub-string prefix* (SP) Bloom filter to determine the position of the query sub-string within a keyword. Based on the different SP Bloom filter matches, the matching keywords are relatively ordered without leaking any other additional information about the similarity of the keywords.

G. Key Contributions

Our key contributions are as follows. (a) We take the first step towards exploring the solution space for the problem of privacy preserving string pattern matching over encrypted data in the symmetric searchable encryption setting. (b) We describe PASStree, a secure searchable index structure, which processes a pattern query in logarithmic time complexity for each possible matching keyword. (c) We devise a relevance ranking algorithm by leveraging the structure of the PASStree to ensure that the most relevant documents of the pattern query are returned in that order. (d) We prove the security of our algorithm under the *semantic security against adaptive chosen keyword attack* IND-CKA. (e) We performed comprehensive experimentation on real-life data sets and show that we achieve 100% accuracy in retrieving all matching keywords with query processing time in the order of few milliseconds for an index over a keyword space of 100000 keywords.

II. Related Work

Secure pattern matching has received much attention in the secure multi-party communication community and several interesting protocols [7]–[10] have been proposed with applications to secure DNA matching and discrete finite-automaton (DFA) evaluation. These solutions address different variations of pattern matching over encrypted data, such as exact pattern matching with index location reporting, wild-card matching, and non-binary hamming distance matching. Although these protocols have extensive pattern matching capabilities, they require expensive computations, such as exponentiations [7], [8], [10], [18], and large communication overhead [9], unsuitable for user oriented cloud computing.

Several symmetric key based efficient techniques [4]–[6], [11]–[14], [16] have been proposed for exact keyword search. However, adopting these solutions for string pattern matching will result in impractical index sizes. Even the recently proposed approach from [15], which is highly storage efficient, cannot solve the string pattern query problem considered in our work. We note that, our approach can be super-imposed over most of these schemes, since the leaf nodes of the PASStree correspond to keywords and therefore, it is possible to incorporate useful features such as dynamic updates [12] to the cloud data.

The work in [17] proposes a technique to search for keywords *similar* to the search keyword on the encrypted data The authors describe an error-correction technique to handle mistakes in user inputs such as typos or incorrect spellings, based on a predefined hamming distance metric. However, the index size stored in this approach grows polynomially with respect to the desired hamming distance. In [13], the authors use *cosine similarity* metric, which is based on the Term-Frequency Inverse-Document-Frequency (TF-IDF) ranking metric, to rank the search results when multiple keywords are searched. However, the focus of [13] is to return the best ranked documents for multiple keyword search and not on pattern matching. In summary, compared to the other symmetric key approaches, our PASStree maintains a smaller index and does not need to predefine the similarity metrics.

III. Pattern Aware Secure Search Tree

In this section, we formally describe the construction of PASStree, our index structure to support pattern queries. Our PASStree description begins with our string pattern matching approach, followed by the structural description of PASStree, the description of the privacy preserving measures for PASStree and finally, the query processing approach.

A. String Pattern Matching

For a given document D, a keyword $w \in D(W)$ is defined as a sequence of characters over a text alphabet $\sum$ and a string pattern $p \in \sum^l$ is defined as a string of l contiguous characters over the same alphabet. Any sub-string of w, denoted by w_{ab}, is a sequence of characters starting from position a and ending at position b, $\forall\ 1 \leq a \leq b \leq |w|$. We denote $S(w) = \cup_{a,b=1}^{a,b=|w|}$, as the set of all sub-strings of the keyword w. We note that, a string pattern p *matches* the keyword w if and only if $p \in S(w)$, *i.e.*, p is a sub-string of w. Next, we denote the set of all sub-strings of all keywords in the document set $\mathbf{D}$ as: $\mathbf{S} = \cup S(w_j)$ where $j = 1$ to $j = |D_i(W)|$ and $i = 1$ to $|\mathbf{D}|$. In our work, we assume that the cardinality of this set, $|\mathbf{S}| = N$, *i.e.*, there are N distinct keywords across all the documents.

Using this formulation, the problem of string pattern matching is reduced to that of membership testing on a set of sub-strings. Thus, our string pattern matching approach is as follows: given a pattern p and document set $\mathbf{D}$, we explicitly generate and store $\mathbf{S}$. In the basic search process, we examine each sub-string in $\mathbf{S}$ and report the m^{th} keyword w_m as a matching keyword, if p is identical to $w_m[a, b]$ where $1 \leq a, b \leq |w_m|$. However, as linear searching is expensive, we describe an efficient tree index structure, the PASStree, which stores the sub-strings of the keywords in a binary tree structure and performs the search in logarithmic complexity.

B. PASStree Structure

To reduce the overhead of matching the pattern query, p, with each keyword w_i, we organize $\mathbf{S}$ in a complete binary tree and narrow the search to only those keywords that possibly match the pattern. We describe PASStree –Pattern Aware Secure Search tree, a highly balanced binary tree, to store $S(w_1), \cdots, S(w_N)$, without revealing any content similarities of the keywords.

PASStree Construction. For an input of N keywords, the PASStree consists of $2N$ distinctly labeled nodes arranged as follows: a root node, $N - 1$ intermediate nodes, and N

leaf nodes Each tree node is associated with a unique Bloom filter [19], which is an efficient storage data structure used for performing set membership testing over large data sets. The Bloom filter is a bit-array of size M where all the bits are initially set to zero. Each Bloom filter is associated with set of q independent hash functions: $h_1, h_2, \cdots, h_q$, where each hash function hashes an input element into the range $[0, M-1]$. Given a set A of elements, to store an element $a \in A$ in the Bloom filter, we hash a using each of the hash functions as follows: $h_1(a), h_2(a), \cdots, h_q(a)$ and set the respective Bloom filter array positions to 1. After storing all elements from A, to check if a query element b belongs to A, the query element is hashed using each of the q hash functions. After hashing, b is declared to belong to A, only if, all the Bloom filter bits in the hashed locations are equal to 1. Note that, this property of Bloom filters is useful for the string pattern matching problem since we are checking if the query string pattern is a member of the set of all sub-strings of keywords in the document set. Now, we state two fundamental properties of a PASStree index structure. First, the Bloom filter associated with each leaf node stores the set of all sub-strings $S(w_i)$ of a unique keyword w_i. Second, at the intermediate tree nodes, the Bloom filters store the union of the elements stored in the respective Bloom filters of the child nodes. Typically, the size of a parent Bloom filter is approximately twice as big as any of its child Bloom filters.

The root node R stores the sub-strings of all the keywords, *i.e.*, the root node stores $\mathbf{S}$, where each sub-string is stored using the standard Bloom filter hashing approach [19]. Now, we create two child nodes for the root node and denote them by the binary tree terminology, as R_{left} and R_{right} child nodes. Next, the set $\mathbf{S}$ is partitioned into two mutually exclusive sub-sets, $\mathbf{S_1}$ and $\mathbf{S_2}$ where each sub-set is associated with one child node, *i.e.*, $\mathbf{S_1}$ is stored in R_{left} and $\mathbf{S_2}$ is stored in R_{right}. The partitioning satisfies the following important conditions: $|R_{left}| = |R_{right}|$ if $|\mathbf{S}|$ is even, and $||R_{left}| - |R_{right}|| = 1$ if $|\mathbf{S}|$ is odd. The partitioning is repeated recursively until each leaf node is associated with a single set $S(w)$ of a distinct keyword w. Based on the balancing conditions, this partitioning approach ensures that the height of the PASStree is $\log N$.

Since the expected output of the pattern query is the set of documents containing the matching keywords, the PASStree incorporates an inverted table index, *i.e.*, for each distinct keyword we maintain a list of document identifiers for documents containing the keyword. For a given keyword w, the document identifier list is denoted by $L(w) = \{D_a, D_b, \cdots, Dv\}$. Given a keyword w, we add a pointer from the corresponding leaf node in the PASStree to $L(w)$, and if any pattern matches this leaf node, the list of identifiers is included in the output.

PASStree Search. To process a query over the PASStree, the approach is as follows. We denote the root of the PASStree by R and R_{left} and R_{right} to denote the logical left and right subtrees of R, respectively. The identifiers of R_{left} and R_{right} are chosen uniformly at random and stored along with the Bloom filter corresponding to R, *i.e.*, as a tuple $\langle R, ID(R_{left}), ID(R_{right})\rangle$ where $ID(X)$ is the identifier of X. For a given pattern query, we first check the R Bloom filter and if a match is found, we proceed further down the PASStree. If R_{left} or R_{right} report a match, the search continues further in that sub-tree and if no Bloom filter reports a match, meaning that no keyword contains this pattern, we stop exploring that sub-tree further. The output of the algorithm is the list of documents that are associated with the leaf nodes An illustrative PASStree is shown in Figure 1.

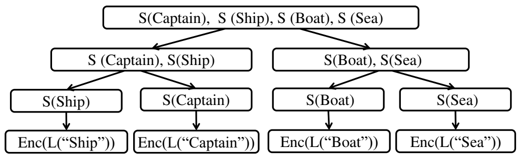


Figure 1. PASStree Example

Theorem 1 (Logarithmic Complexity): For a given query string p, the PASStree search algorithm finds all matching leaf nodes in a complexity of $O(E \log N)$ where N is the total number of leaf nodes in the PASStree and E is the number of leaf nodes matching p.

Proof. First, we show the search time complexity of a pattern p, which matches only one leaf node. At each level from the root node, the search algorithm checks the nodes R_{left} and R_{right}. Since the pattern query matches only one leaf node, the search algorithm proceeds along the sub-tree where the match is found and does not explore the other sub-tree. Therefore, the search algorithm performs at most two Bloom filter verifications at each level in the PASStree until the leaf node, which translates to at most $2 \log N$ complexity.

Next, we consider the case where the pattern p matches at most E leaf nodes. In the worst case scenario, the E leaf nodes will be found in E different sub-trees Given that the cost of verification along each independent sub-tree is $2 \log N$, the aggregated complexity of searching for pattern p is $2E \log N$. If E is of the order of N, say, $\frac{N}{c}$ for some c, the complexity of searching in PASStree is $O(N \log N)$. This result proves that the PASStree achieves polynomial time complexity in worst-case and achieves logarithmic complexity if a pattern matches only a small number of keywords. □

C. Preserving Privacy of Bloom Filters

As the Bloom filter data is available on the cloud server, an adversary can make some inferences regarding the data directly or indirectly. For instance, the parameters of the Bloom filter, the hash functions and the bit-array size, need to made public since the cloud server needs to perform the set membership operation. The Bloom filter size M-bits depends on the number of distinct keywords denoted by $|D_i(W)|$ for a document D_i and the number of sub-strings in each keyword. Using this information, an adversary can perform brute force dictionary attacks and determine the content or the number of elements stored in the Bloom filter. Furthermore, the common bit positions between two different Bloom filters can be used to infer the common elements stored in two different Bloom filters. To address these concerns, we describe our privacy preserving approach to protect the content of the Bloom filters.

Secure Trapdoor Generation. We assume that the data owner and the authorized users share a t-bit common secret master key, $K \in \{0,1\}^t$, and agree on a common secure one-way hash function $\mathcal{H}$, such as SHA-2. The data owner computes q secret keys as follows: $K_1 = K||1, K_2 = K||2, \cdots, K_q = K||q$. Now, for each unique sub-string $w_{ab} \in S(w)$ of a keyword $w \in D_i$, we compute q secure one-way hashes as follows: $\mathcal{H}(K_1, w_{ab}), \mathcal{H}(K_2, w_{ab}), \cdots, \mathcal{H}(K_q, w_{ab})$. This approach ensures the privacy of the sub-string w_{ab}, as it is easy to compute a one-way hash, given K and w_{ab}, but it is computationally infeasible to determine w_{ab}, given these one-way hashes.

Sub-string Randomization. Storing the one-way hashes of the sub-strings into various Bloom filters in the PASStree does not prevent passive inference, *i.e.*, an adversary can examine the Bloom filters for common locations and determine the common elements directly. Therefore, we use the randomizing approach from [4] while storing an identical hash value in different Bloom filters. We note that, each PASStree tree node has a distinct label, $\mathcal{L}_z$. Using this label, we compute the q Bloom filter hashes as follows:$\mathcal{H}(\mathcal{L}_z, \mathcal{H}(K_1, w_{ab}))\%M$, $\mathcal{H}(\mathcal{L}_z, \mathcal{H}(K_2, w_{ab}))\%M, \cdots, \mathcal{H}(\mathcal{L}_z, \mathcal{H}(K_q, w_{ab}))\%M$. Since the PASStree node labels are chosen uniformly at random, the sub-string w_{ab}, will hash into different bit-positions across two Bloom filters of two different PASStree nodes, thereby, eliminating correlation based attacks.

Blinding the Bloom Filters. Finally, we blind each Bloom filter [4], at the same distance from the PASStree root, by inserting $(|W| - |T|) * V_r$ random 1s into the different Bloom locations. Here, $|T|$ is the number of distinct sub-strings in the Bloom filter, $|W|$ is the maximal number of sub-strings stored in a Bloom filter at the same level and V_r is a random integer constant. This ensures that all Bloom filters at the same distance from the PASStree root node contain the same number of 1s regardless of the number of sub-strings stored. This blinding is necessary to provide security against the passive inference of common elements across two Bloom filters.

D. Query Trapdoor Generation and Processing

The user specifies a string $p \in \sum$ as a pattern query and generates the trapdoor, T_p as follows: $\mathcal{H}(K_1, p), \mathcal{H}(K_2, p), \cdots, \mathcal{H}(K_q, p)$. The cloud server executes the search algorithm, as described in Section III-B, starting from the root as follows: $\mathcal{H}(\mathcal{L}_R, \mathcal{H}(K_1, p))\%M$, $\mathcal{H}(\mathcal{L}_R, \mathcal{H}(K_2, p))\%M, \cdots, \mathcal{H}(\mathcal{L}_R, \mathcal{H}(K_q, p))\%M$ where L_R is the label of the root Bloom filter. If all the hashed locations are set to 1, the p is a sub-string of one or more keywords and the search proceeds along the children of the root node. Proceeding thus, the leaf nodes matching the trapdoor are declared to matching the query string pattern and the corresponding list of the documents are retrieved.

IV. PASSTREE+

The PASStree construction does not take advantage of the similarity of different keywords and hence, might result in the worst case search behavior in many scenarios. In this section, we first describe the technical challenge involved in search optimization and present a novel heuristic algorithm towards this challenge.

A. Challenge in Search Optimization

During the PASStree search, if a pattern is found in both the sub-trees, T_{left} and T_{right}, the search algorithm proceeds along both the paths and results in maximum possible paths being explored along the length of the PASStree. This scenario arises due to the unstructured partitioning of the keyword set at a PASStree node, $\mathbf{S_a}$ into two keyword sets $\mathbf{S_{aa}}$ and $\mathbf{S_{bb}}$ in such a way that both the sets might end up sharing many common sub-strings. Hence, it is desirable to achieve PASStree node partitioning while minimizing $\mathbf{S_{aa}} \cap \mathbf{S_{bb}}$, *i.e.*, the number of common sub-strings across two partitions should be minimal while satisfying the constraint: $||\mathbf{S_{aa}}| - |\mathbf{S_{bb}}|| \leq 1$. This problem is a variant of the well-known set partitioning problem and can be shown to be NP-hard in a straightforward manner and can be solved using a greedy heuristic. The key intuition to improving the search efficiency of the PASStree is to group keywords, matching similar query sting patterns, into common sub-tree locations.

B. Optimizing PASStree

To overcome the challenges in search optimization, we describe the construction of PASStree+, an enhanced version of PASStree, which uses a novel heuristic algorithm for the partitioning problem to optimize the search efficiency. First, our heuristic algorithm computes the similarity of the keyword pairs using a similarity metric that not only takes into account the pattern similarity of the keyword pairs, but also the distribution of the keyword pairs across the document set. Second, using the similarity coefficients as keyword weights, our algorithm uses a scalable clustering approach to partition the keyword set.

Keyword Pair Similarity Estimation. We make an important observation that, if two keywords share many patterns then it would be desirable that these two keywords are placed in the same partition, because if the user searches for a pattern common to both the keywords, then the PASStree exploration will be focused only along this partition. Therefore, our approach to improving the search efficiency is to arrange two or more keywords in the same partition by measuring the number of sub-strings they have in common. For instance, given two keywords w_1 and w_2, where $S(w_1) \in \mathbf{S_{aa}}$ and where $S(w_2) \in \mathbf{S_{bb}}$, we can re-arrange them into the same partition, $\mathbf{S_{aa}}$ or $\mathbf{S_{bb}}$, if they share many common sub-strings. We quantify this metric, using the Jaccard similarity coefficient technique [20], as follows:

$$SS_c = \frac{|S(w_1) \cap S(w_2)|}{|S(w_1) \cup S(w_2)|} \tag{1}$$

where SS_c stands for *sub-string similarity co-efficient*. Since this approach does not group the keywords based on a lexical ordering, it does not violate the privacy of the content in the generated PASStree.

But, using only this approach might create partitions purely on a syntactic basis and not on the semantic relation between the keywords. Therefore, it would be desirable to group together common patterns that occur within the same document, as it lead to more relevant documents with respect to the pattern query. For instance, if two keywords, "Shipper" and "Shipment", with common pattern "Ship", are in the same document, then this document is probably most relevant to this pattern. Based on this, we identify two additional metrics for grouping keywords: (a) PS_c *phrase similarity co-efficient*, which measures the fraction of documents in which the two keywords occur as a *phrase*, *i.e.*, one after another, and (b) DS_c *document similarity co-efficient*, which measures the fraction of documents in which the two keywords occur within the same document, but not as a phrase. These two metrics are computed using Jaccard similarity technique as follows. The first metric PS_c is given by following equation: $PS_c = \frac{|L(w_1 \rightarrow w_2)|+|L(w_2 \rightarrow w_1)|}{|L(w_1 \cap \not{w}_2)|+|L(w_2 \cap \not{w}_1)|+|L(w_1 \cap w_2)|}$ where $L(w_1 \rightarrow w_2)$ indicates the list of documents in which w_1 and w_2 occur as a phrase; $|L(w_1 \cap \ /w_2)|$ is the number of documents containing only w_1 but not w_2 and so on. The second metric DS_c is as follows: $DS_c = \frac{|L(w_1 \cap w_2)|}{|L(w_1 \cap \not{w}_2)|+|L(w_2 \cap \not{w}_1)|+|L(w_1 \cap w_2)|}$. Based on these metrics, we quantify the similarity coefficient $S_c(w_1, w_2)$ of a keyword pair, w_1 and w_2, as the sum of the individual Jaccard similarity coefficients: $S_c(w_1, w_2) = SS_c + PS_c + DS_c$

Partitioning Using Clustering. We use CLARA [21], a well known clustering algorithm, to partition a keyword set based on the keyword pair similarity. CLARA clusters the keywords around a representative keyword, called a medoid, MED, such that all the keywords share a high S_c with the medoid of a cluster. The medoids are chosen from a sample space of randomly selected keywords from the complete keyword set. Through thorough experimental analysis, [21] suggests that 5 sample spaces of size $40 + 2k$ give satisfactory results, where k is the number of required clusters. In our partitioning problem, we need 2 clusters, thus k=2 and the corresponding sample space is of size 44. Finally, as suggested by [21], to get the best possible medoids, we perform 5 iterations and balance the clusters to contain equal number of items.

V. Ranking Search Results

As a query string pattern can match several keywords in the document set, it is necessary to determine the relative importance of a matching keyword to the query string. We define *ranking* as an ordering of the matching leaf nodes with respect to a query string This implies that for two different string patterns matching the same set of leaf nodes, the ranking of the leaf nodes will be likely to be distinct. Note that, our definition of *ranking* is different from the conventional ranking defined in works such as [13]. *Ranking Heuristic.* We use a simple metric to quantify the relevance of a keyword to a given pattern: the position of the first occurrence of the pattern in the keyword determines the *rank* of the keyword with respect to the pattern. We have adopted this strategy as it is used in popular search engines such as Google$^{(tm)}$ and there may be other suitable alternative strategies depending on the application domain. Formally, if the characters in a keyword w are numbered as $1, 2, \cdots, |w|$, and if a given query sub-string p begins at the j^{th} position of w, then the rank of w is j with respect to p where a smaller j means a higher rank. For example, for a query string *Ship*, a set of matching keywords $Shipment$, $Shipper$, $Worship$, will be ranked 1, 1 and 2 respectively, based on the position of $Ship$ in these keywords. Therefore, to return the most relevant documents to the user, the matching keywords, *i.e.*, the leaf nodes of the PASStree are ranked high to low based on the ranks of the respective keywords for the pattern string query.

A. Recording Matching Positions

To determine the position of the matching string pattern in a keyword, we use a heuristic approach leveraging the structure of the PASStree and store additional information in the PASStree to record the matching positions. For each PASStree node, we store an auxiliary Bloom filter, called the *sub-string prefix* (SP) Bloom filter. For uniformity, we denote the regular Bloom filter as the R Bloom filter. For a PASStree node at a distance d from the leaf nodes, the corresponding SP Bloom filter stores *all* possible prefixes of the node's keywords for the sub-strings at the d^{th} positions of the keywords. For instance, at a leaf node, the SP Bloom filter stores all the prefixes of the keyword, corresponding to sub-strings at 1^{st} position, *e.g.*, given keyword *"Ship"*, the SP Bloom filter stores, *"S, Sh, Shi, Ship"*. Next, the SP Bloom filter at the parent node of the leaf nodes stores all the prefixes corresponding to the sub-strings found at the 2^{nd} position in the keywords and so on. At the child nodes of the PASStree root, the SP Bloom filter stores all the prefixes corresponding to any substrings still remaining. The storage technique is same as that described in Section III, where each SP Bloom filter has a distinct identifier, and the same set of cryptographic keys are used as done within the regular Bloom filters of the PASStree nodes.

B. Ranking Algorithm

Our ranking approach assigns ranks in ascending order, *i.e.*, 1 is highest and so on. The key intuition of the ranking algorithm is as follows: if a query string, say "p", matches an SP filter of a leaf node, then this leaf node receives the highest rank of 1. If an intermediate SP filter node matches the query string, then the ranking of the matching leaf nodes in this node's sub-tree is decided by the distance of the SP filter node from the leaf node, *i.e.*, the farther the SP node, the lower its rank. The detailed approach is described as follows.

At each PASStree node, the search algorithm performs two checks: once in the regular Bloom filter and another in the SP Bloom filter. If there is a match in SP Bloom filter, then the height d_i of the matching PASStree node and the identifier BF_{id} of the PASStree node Bloom filter are recorded as: $M_i =< d_i, BF_{id} >$ are recorded in the set $R = R \cup M_i$ where R sorts the tuples in ascending order of the d_i values. When the search terminates, the ranking algorithm, chooses the first $M_i =< d_i, BF_{id} >$ from R, and assigns the highest rank to all the matching leaf nodes within the subtree of the PASStree node corresponding to BF_{id}, and moves to the next node in R. Proceeding in this manner, all the matching leaf nodes in the PASStree are arranged in the ranked order.

VI. Security Analysis

A. Security Model

To achieve IND-CKA security, our PASStree structure uses *keyed* one-way hash functions as pseudo-random functions whose output cannot be distinguished from a truly random function with non-negligible probability [22]. We have used $\mathtt{SHA-2}$ for our scheme as the pseudo-random function $\mathcal{H}$ and $\mathtt{AES}$ as the encryption algorithm Enc for the documents. From [6], [22], in the simulation based security model, a searchable symmetric encryption (SSE) scheme is IND-CKA secure if any probabilistic polynomial-time adversary cannot distinguish between the trapdoors generated by a real index using pseudo-random functions and a simulated index using truly random functions, with non-negligible probability. The important part of our proof is the construction of a polynomial time simulator, which can simulate the PASStree and hence, show the IND-CKA security conclusively. A probabilistic polynomial-time adversary interacts with the simulator as well as the real index and is challenged to distinguish between the results of the two indexes with non-negligible probability. We consider a *non-adaptive adversary*, *i.e.*, prior to the simulation game, the adversary is not allowed to see the history of any search results or the PASStree.

B. Security Proof

Without loss of generality, the PASStree can be viewed as a list of Bloom filters, where each Bloom filter stores the sub-strings corresponding to a distinct keyword and matches different string patterns. The leaf node points to a list of document identifiers containing the keyword and the list can be encrypted as well using well known techniques from [6] and [23]. Therefore, proving the PASStree IND-CKA secure is equivalent to proving that each Bloom filter is IND-CKA secure with the following properties: (a) the Bloom filter bits do not reveal the content of the stored strings and (b) any two Bloom filters storing the same number of strings, with possibly overlapping strings, are indistinguishable to any probabilistic polynomial-time adversary. Given that the Bloom filter identifiers are distinct and random, the same pattern is stored in different locations across different Bloom filters. We use the key length s as the security parameter in following discussion.

History: H_q. Let $\mathbf{D} = \{D_1, D_2, \cdots, D_n\}$ denote the set of document identifiers where D_i denotes the i^{th} document. The history H_q is defined as $H_q = \{\mathbf{D}, p_1, p_2, \cdots, p_q\}$, where the set $\mathbf{D}$ consists of document identifiers matching one or more query string patterns p_1 to p_q. An important requirement is that q must be polynomial in the security parameter s, the key size, in order for the adversary to be polynomially bounded.

Adversary View: A_v. This is the view of the adversary of a history H_q. Each query pattern, p_i generates a pseudo-random trapdoor T_{p_i} using the secret key $K \in \{0,1\}^s$. The adversary view is: the set of trapdoors corresponding to the query strings denoted by $\mathbf{T}$, the secure index $\mathcal{I}$ for $\mathbf{D}$ and the set of the encrypted documents, $Enc_K(D)$=$\{Enc_K(D_1), \cdots, Enc_K(D_n)\}$, corresponding to the returned document identifiers. Formally, $A_v(H_q) = \{\mathbf{T}; \mathcal{I}; Enc_K(D)\}$.

Result Trace. This is defined as the *access* and *search* patterns observed by the adversary after $\mathbf{T}$ is processed on the encrypted index $\mathcal{I}$. The access pattern is the set of matching document identifiers, $M(T)$=$\{ m(T_{p_1}), \cdots, m(T_{p_q})\}$ where $m(T_{p_i})$ denotes the set of matching document identifiers for trapdoor T_{p_i}. The search pattern is a symmetric binary matrix Π_T defined over T, such that, $\Pi_T[p,q] = 1$ if $T_p = T_q$, for, $1 \leq p, q \leq \sigma^{|T_i|}$. We denote the matching result trace over H_q as: $M_{(H_q)} = \{M(T), \Pi_T[p,q]\}$.

Theorem 2 (IND-CKA Security Proof): The PASStree scheme is IND-CKA secure under a pseudo-random function f and the symmetric encryption algorithm Enc.

Proof. We show that given a real matching result trace $M_{(H_q)}$, it is possible to construct a polynomial time simulator $\mathcal{S} = \{S_0, S_q\}$ simulating an adversary's view with non-negligible probability. We denote the simulated index as $\mathcal{I}^*$, the simulated encrypted documents as, $Enc_K(D^*)$ and the trapdoors as $\mathbf{T}^*$. Recall that, each Bloom filter matches a distinct set of trapdoors, which are visible in the result trace of the query. Let ID_j denote the unique identifier of a Bloom filter. The expected result of the simulator is to output trapdoors based on the chosen query string history submitted by the adversary. The adversary should not be able distinguish between these trapdoors and the trapdoors generated by a real PASStree with non-negligible probability.

Step 1. Index Simulation To simulate the index $\mathcal{I}^*$, we generate $2N$ random identifiers corresponding to the number of Bloom filters in the index and associate a $depth$ label with each string to denote its distance from the root. We generate random strings $Enc_K(D^*)$, such that each simulated string has the same size as an original encrypted document in $Enc_K(D)$ and $|Enc_K(D^*)| = |Enc_K(D)|$.

Step 2. Simulator State S_0 For H_q, where $q = 0$, we denote the simulator state by S_0. We construct the adversary view as follows: $A_v^*(H_0) = \{Enc_K(D^*), \mathcal{I}^*, T^*\}$, where T^* denotes the set of trapdoors. To generate T^*, each document identifier $Enc_K(D^*)$ corresponds to a set of matching trapdoors. The length of each trapdoor is given by a pseudo-random function and the maximum possible number of trapdoors matching an identifier is given by the average maximum number, denoted by δ, of sub-strings of a keyword. Therefore, we generate $(\delta + 1) * |Enc_K(D^*)|$ random trapdoors and uniformly associate at most $\delta + 1$ trapdoors for each data item in $Enc_K(D^*)$. Note that, some trapdoors might repeat, which is desirable as two documents might match the same trapdoor. This distribution is consistent with the trapdoor distribution in the original index $\mathcal{I}$, *i.e.*, this simulated index satisfies all the structural properties of a real PASStree index. Now, given that $\mathtt{SHA}-2$ is pseudo-random and the probability of trapdoor distribution is uniform, the index $\mathcal{I}^*$ is indistinguishable by any probabilistic polynomial time adversary.

Step 3. Simulator State S_q For H_q where $q \geq 1$, we denote the simulator state by S_q. The simulator constructs the adversary view as follows: $A_v^*(H_q) = \{Enc_K(D^*), \mathcal{I}^*, T^*, T_q\}$ where T_q are trapdoors corresponding to the query trace. Let p be the number of document identifiers in the trace. To construct I^*, given M_{H_q}, we construct the set of matching document identifiers for each trapdoor. For each document identifier in the trace, $Enc_K(D_p)$, the simulator associates the corresponding real trapdoor from $M(T_i)$ and if more than one trapdoor matches the document identifier, then the simulator generates a union of the trapdoors. As $q < |\mathbf{D}|$, the simulator generates $1 \leq i \leq |\mathbf{D}| - q + 1$ random strings, $Enc_K^*(D_i)$ of size $|Enc_K(D)|$ each and associates up to $\delta + 1$ trapdoors uniformly, as done in Step 2, ensuring that these strings do not match any strings from $M(T_i)$.

However, this construction cannot handle the cases where an adversary generates sub-strings from a single keyword and submits them to the history. For instance, the trapdoors corresponding to $Stop, top, op$ and $flop$ will result a set of common document identifiers as some of these patterns will be found in the same documents. Therefore, in such cases, the adversary expects to see some of the document identifiers to repeat within the result trace and if this does not happen, the adversary will be able to distinguish between a real and simulated index. To address this scenario, we take the document identifiers from the real trace and for each of the random Bloom filter identifiers, we associate a unique subset of these identifiers. This ensures that given any q query trapdoors, the intersection of the document identifiers induced due to this q query history is non-empty and therefore, indistinguishable from a real index. The simulator maintains an auxiliary state $\mathcal{ST}_q$ to remember the association between the trapdoors and the matching document identifiers. The simulator outputs: $\{Enc_K(D^*), \mathcal{I}^*, T^*, T_q\}$. Since all the steps performed by the simulator are polynomial and hence, the simulator runs in polynomial time complexity.

Now, if a probabilistic polynomial time adversary issues a query string pattern over any document identifier matching the set M_{H_q} the simulator gives the correct trapdoors. For any other query string pattern, the trapdoors given by simulator are indistinguishable due to pseudo-random function. Finally, since each Bloom filter contains sufficient blinding, our scheme is proven secure under the IND-CKA model. □

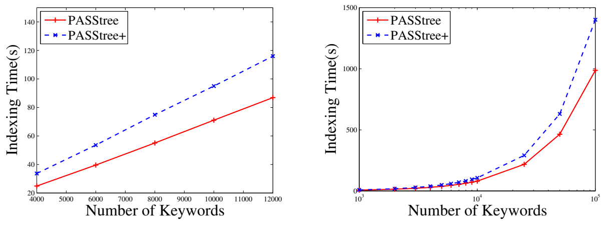
VII. Performance Evaluation

A. Experimental Methodology

To evaluate the performance of our approach, we considered three key parameters to design the experimental configurations: the data size, the PASStree type, and the pattern query result size. Based on these parameters we have done a thorough experimental evaluation of the index size, index construction time, the query processing time and the ranking precision.

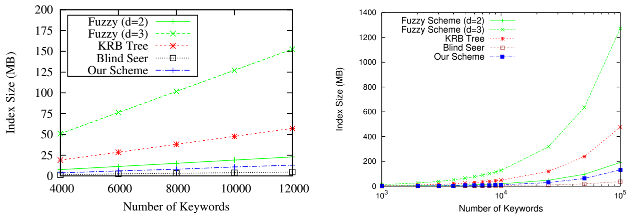
1) Data Sets: The number of keywords is the most important factor in the PASStree evaluation and we select document collections containing different sized keywords set. We chose two real-life data sets: the Wikipedia archival pages from year 2010 and the *Enron* email dataset, containing mails of Enron employees over a certain period. We denote the Wikipedia data set by $WIKI$ and the Enron data set by $ENRON$. The $WIKI$ data set consists of 10000 documents, chosen out of a 10 million plus corpus, where each document contains an average of 100 distinct keywords, not counting the general stopwords such as, "a, an, the, there" and so on. We chose distinct collections of $WIKI$ documents containing, on an average, $1000, 5000, 25000, 50000$ and 100000 keywords, and averaged the results over 5 different collections for each keyword size configuration. For instance, to get one instance of 1000 keywords data set, we chose up to 100-150 distinct documents and similarly, for the other 4 instances of 1000 keywords data set, we chose four more distinct document collections. We ran the PASStree and PASStree+ algorithms on each of these five different 1000 keywords data sets and averaged the results. Similar experiments were repeated for the other keyword data set sizes as well.

The $ENRON$ email data set consisted of 10000 documents, chosen out of 0.6 million corpus, where each docu-



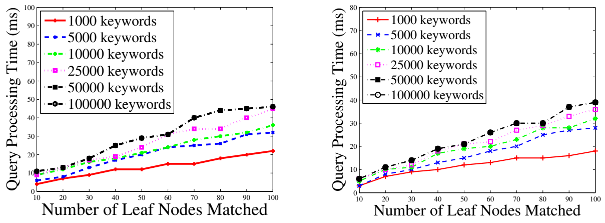
(a) ENRON (b) WIKI

Figure 2. Index Construction Time



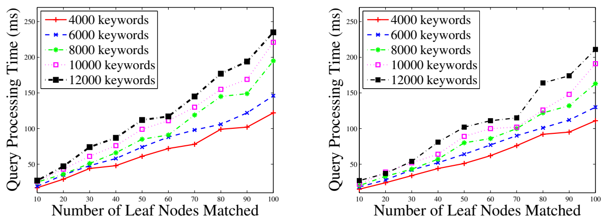
(a) ENRON (b) WIKI

Figure 3. Index Size



(a) PASStree (b) PASStree+

Figure 4. Query Time for WIKI



(a) PASStree (b) PASStree+

Figure 5. Query Time for ENRON

ment contained an average of 10 distinct keywords, besides the stop words. Each $ENRON$ document corresponds to an email between a $Sender$ and a $Receiver$ with the email header and content exchanged. For the $ENRON$ data set, we considered different document collections containing, $4000, 6000, 8000, 10000$ and 12000, distinct keywords and averaged the results over 5 different collections for each configuration. The keyword data set configuration is similar to the $WIKI$ data set configuration.

2) Implementation Details: The PASStree was implemented in C++ and the experiments were conducted on desktop PC running Linux Ubuntu 12.10 with $4GB$ memory and *3.3GHz Intel(R) Core(TM) i3-2120k* processor. To encrypt the keywords we use the Advanced Encryption Standard (AES) algorithm with a 128-bit master key and $HMAC - SHA2$ as the secure hash function for the Bloom filter encoding. We set the Bloom filter parameter, $M/N = 10$, where M is the Bloom filter size and N is the number of elements and the number of Bloom filter hash functions: $q = 7$. We chose $t = 256$ as the bit-length for each of the K_i where $1 \leq i \leq q$ keys used to compute the trapdoor hashes over an encrypted pattern query.

3) Query Types: For the $WIKI$ data set, we considered string pattern queries where the user retrieves documents containing keywords that match a sub-string. We denote these queries as $sub - string$ queries. For the $ENRON$ data set, we considered queries that wish to retrieve emails between a particular $SENDER$ to a particular receiver $RECEIVER$. Since our PASStree approach includes the co-occurrence of the two keywords as one of the similarity metrics, this experiment is useful to determine whether PASStree can achieve multi-keyword phrase search effectively. We denote these queries as $SENDER/RECEIVER$ queries. For $SENDER/RECEIVER$ queries in Enron data set, the individual keywords from sender and receiver are extracted. An example $SENDER/RECEIVER$ query is: *"From: Grant Colleean To: Market Status"*, which is converted to a set of 4 keywords query: $Grant$, $Colleean$, $Market$ and $Status$. The query is performed on individual keywords and the results are accumulated.

Next, the query result size, *i.e.*, number of matches for a query string, is an important factor affecting the performance of the PASStree and therefore, we chose the $sub - string$ and $SENDER/RECEIVER$ queries, which have different query result sizes ranging from: $10, 20, \cdots, 100$. We averaged each of the experiments over 10 different trials, *e.g.*, choosing 10 distinct queries resulting in 10 different matching leaf nodes and recording the average of the metrics involved.

B. PASStree Construction and Size

Our experimental evaluation shows that the PASStree construction time is small for large data sets and the index size is either comparable or smaller to existing works. Figures 2(a) and 2(b) show the construction time for PASStree and PASStree+ over different sized keyword sets, and as expected, PASStree construction is faster. For reasonably large sized sets of 1000 to 10000 keywords, the construction time is less than 100 seconds, and for very large keyword sizes, > 10000, the construction time is between 100 seconds to 1500 seconds.

To compare the size of PASStree PASStree+, we chose several existing privacy preserving keyword search schemes. The scheme in [15] is denoted as *Blind Seer* wherein this scheme uses a Bloom tree to index documents instead of keywords. This scheme is highly space efficient but can only support keyword queries. The scheme in [17] supports similarity based search using hamming distance as similarity metric and we denote this scheme by *Fuzzy d=x* where x is the chosen hamming distance. The index sizes in this scheme are very high and cannot support some queries like "*Immun*" to match keywords like "*Immunization*", because the hamming distance is $d = 7$ and for this value of d the index size is prohibitive. Finally, we compare with the scheme in [24] and denote it by *KRB Tree*, which supports dynamic keyword search. Figure 3(a) and 3(b), show that the PASStree size is much smaller than the *Fuzzy* scheme and comparable to the other keyword only search schemes. These results show that PASStree sizes are very reasonable even for large keyword spaces and therefore, is suitable for practical deployment.

C. Query Processing Speed and Accuracy

Our experiments show that the pattern query matching achieves 100% *accuracy and the query processing time is very efficient over large data sets.* Figures 4 and 5 show the query processing times over PASStree and PASStree+ for different keyword sets for various query result sizes. While PASStree+ is faster than PASStree, as is expected, both the structures execute large queries within few 10s of milliseconds, which is very fast considering the data set sizes.

D. Ranking Precision

Our ranking approach is very effective in returning the most relevant matching documents. For the sake of experiments, we maintained a ranked list of documents at the leaf nodes and compared the relative ranking of the PASStree results with this ranked list. We measured the *ranking precision* as L'/L where L' denotes the number of top-ranked documents returned in the PASStree results and L denotes the actual number of top-ranked documents. The experiments show that, for the $WIKI$ data, the ranking precision is between 90% to 100%, whereas, for the $ENRON$ data, the values are between 90% to 100% on an average, with an odd value falling to 70%.

VIII. Conclusion and Future Work

In this work, we presented the first approach towards privacy preserving string pattern matching in cloud computing. We described PASStree, a privacy preserving pattern matching index structure along with novel algorithms to solve the various challenges in this problem. Our PASStree structure can process pattern queries in a fast and efficient manner over several thousands of keywords. The results of our paper demonstrate the need to further explore the rich problem space of privacy preserving pattern matching in cloud computing. We also demonstrated strong security guarantees, which shows that our approach can be deployed in practical systems. The future scope of this work lies in exploring more expressive pattern querying mechanisms for user friendly cloud computing data applications.

References

[1] R. Grossi and J. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 378–407, 2005.

[2] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," ser. FOCS. IEEE Computer Society, 2000.

[3] R. Canetti, U. Feige, O. Goldreich, and M. Naor, "Adaptively secure multi-party computation," in *Proc. 28th ACM STOC*, 1996, pp. 639–648.

[4] Eujin-Goh, "Secure indexes," 2004, stanford University Technical Report.

[5] D. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *IEEE S&P Symposium*, 2000.

[6] R. Curtmola, G. A. J., S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, pp. 895–934, 2011.

[7] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik, "Privacy preserving error resilient dna searching through oblivious automata," ser. ACM CCS, 2007, pp. 519–528.

[8] J. Katz and L. Malka, "Secure text processing with applications to private dna matching," ser. ACM CCS, 2010, pp. 485–492.

[9] P. Mohassel, S. Niksefat, S. Sadeghian, and B. Sadeghiyan, "An efficient protocol for oblivious dfa evaluation and applications," ser. CT-RSA, 2012, pp. 398–415.

[10] J. Baron, K. El Defrawy, K. Minkovich, R. Ostrovsky, and E. Tressler, "5pm: Secure pattern matching," ser. SCN. Springer-Verlag, 2012, pp. 222–240.

[11] Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Third Int. Conf. on Applied Cryptography and Network Security*, 2005.

[12] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *ACM CCS*, 2012.

[13] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li, "Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking," ser. ASIA CCS. ACM, 2013, pp. 71–82.

[14] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 222–233, 2014.

[15] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin, "Blind seer: A scalable private dbms," in *IEEE S&P Symposium*, 2014.

[16] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *ISOC NDSS*, 2014.

[17] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, "Fuzzy keyword search over encrypted data in cloud computing," in *IEEE INFOCOM*, 2010, pp. 441–445.

[18] C. Hazay and T. Toft, "Computationally secure pattern matching in the presence of malicious adversaries." *Journal of Cryptology*, pp. 358–395, 2014.

[19] B. H. Bloom, "Spacetime tradeoffs in in hash coding with allowable errors," *Commun. of the ACM*, vol. 13, pp. 422–426, 1970.

[20] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[21] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Series in Probability and Statistics, 1990.

[22] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Chapman & Hall/CRC Press, 2007.

[23] X. Zou, Y.-S. Dai, and E. Bertino, "A practical and flexible key management mechanism for trusted collaborative computing," in *IEEE INFOCOM*, 2008.

[24] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography*, ser. Lecture Notes in Computer Science, vol. 7859, 2013, pp. 258–274.