Privacy Preserving String Pattern Matching on Outsourced Data

Thesis submitted in partial fulfillment
of the requirements for the degree of

MS By Research
in
CSE

by

Bargav Jayaraman
201207509
bargav.jayaraman@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
May 2015

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Privacy Preserving String Pattern Matching on Outsourced Data" by Bargav Jayaraman, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Adviser: Dr. Bruhadeshwar Bezawada

To my Family and Friends

Acknowledgements

I would like to acknowledge the efforts of the people who have helped me during my research and without whom, this thesis would have not been possible.

Firstly, I would like to express my sincere gratitude to my thesis advisor Dr. Bruhadeshwar Bezawada for teaching me the foundations of research which greatly motivated me to pursue research, not just limited to my Masters, but also beyond. This thesis would not have been possible without his tremendous patience, motivation, constructive criticism and immense knowledge. He has not only taught me how to identify and approach a problem, but also taught some basic skills which are invaluable for a researcher. He has been like a beacon for me throughout my research at every stage. I take this opportunity to wholeheartedly thank him for his guidance in my research.

I would like to thank Prof. Kamalakar Karlapalem for teaching me *Data Warehousing and Data Mining*, which directly helped me in my research. His unique way of teaching the course from a research perspective further shaped my interests towards research. He has been a great inspiration to me. I would also like to express my gratitude for allowing me to participate in COMAD 2014, which was an experience of a lifetime. I would also like to thank Dr. Vikram Pudi for providing me an opportunity to work alongside with him as his TA for *Data Warehousing and Data Mining* and also as an RA for the same course in Enhance Edu. My interactions with him has always been joyful.

I am thankful to all the faculty members of CSTAR - Dr. Kannan Srinathan, Dr. Kishore Kothapalli, Dr. Ashok Kumar Das and Dr. Sanjay Rawat for providing a wonderful research center. I especially want to thank Dr. Ashok Kumar Das for not just teaching me courses like *Principles of Information Security* and *System and Network Security*, but also for providing me a wonderful opportunity to be TA for *Principles of Information Security* which gave me even more insights into the fundamentals of security than when I was just a student in the course. He also gave me some insights into the ongoing research in wireless sensor networks, which stirred an immense interest in me.

I would like to thank Dean R&D, Prof. Vasudeva Varma and Director, Prof. P.J. Narayanan for providing a wonderful research atmosphere.

I am thankful to all my friends who made my life in IIIT a joyful and amazing experience. I would like to give a special mention to my buddies Prathyush and Pramod who were more like my brothers.

Finally, I wholeheartedly thank my family for being ever supportive. I dedicate my thesis to them.

Abstract

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. Thus, the data needs to be encrypted prior to outsourcing.

However, the encryption is a major hindrance to perform regular data access operations, such as searching for documents containing specific keywords or patterns. 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. While there have been many works on efficiently solving the problem of privacy preserving single keyword and multi-keyword search on outsourced data in cloud computing, both in public key and symmetric key domains, the problem of privacy preserving string pattern search has only been solved in public key domain and that too is not efficient enough for adopting to practical scenarios like cloud. We note that the solution to this problem is unexplored in symmetric key domain.

In this work, we present the first ever symmetric key based approach to support privacy preserving string pattern matching on outsourced data in cloud computing. String pattern matching functionality can find occurrence of a string pattern *anywhere* within a keyword. 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. We have performed *prefix* and *sub-string* pattern search on Wikipedia dataset, and performed *multi-keyword phrase search* of sender/receiver queries on Enron dataset. Furthermore, we also describe a relevance ranking algorithm to return the most relevant documents to the user based on the query pattern. Our ranking algorithm achieves 90%+ above precision in ranking the returned documents.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Emergence of Cloud Technologies have revolutionized the landscape of present day Information-Centric World. Many cloud service providers are coming to the forefront with cost effective and user friendly cloud services. Some of the popular cloud service providers are Amazon S3, Microsoft Azure and Google App Engine. The cloud services they offer are mainly of two types: *outsourcing computation* and *outsourcing storage*.

Outsourcing computation is a service where a client having low computation resources can outsource its computation intensive operations to cloud server and get the results. This type of service is widely used by many research universities, which may not be able to invest a lot on the infrastructure.

Outsourcing storage is a more common type of cloud service where a data owner can outsource the data storage to cloud server and can later retrieve the data. Various individual users and organizations use this service. For example, Dropbox provides 5 GB and Google provides upto 15 GB of storage space on their cloud servers for each user. They also provide additional storage space at reasonable prices.

With the lucrative option of pay-as-you-use, many individual users and organizations are outsourcing their data storage to third party cloud service providers. A data owner can outsource his data to the cloud server and can query on the outsourced data or authenticate a client to perform query at a later point of time. This work focuses on the outsourcing storage cloud service.

Various domains where searching is performed on outsourced data are:

- *Search Engine*, where a document collection is outsourced to cloud server and client can retrieve documents which contain the query keywords.
- *Personalized Medication*, where client's medical record is outsourced to hospital's server and an authorized doctor can perform secure searching on client's medical record for diagnosis.
- *Email Server*, where a collection of personal emails is outsourced to email server and client can retrieve relevant emails based on the content of the mail or sender/receiver names or email IDs.
- *Crime Investigation*, where the Interpol criminal database acts as the server and clients are the authorized crime investigation agencies like local police departments.

The data that is being outsourced might be sensitive like patient's medical records, financial data, etc. Hence, outsourcing plain data raises some privacy issues. The data owner may not afford to leak the data to the cloud server or any unauthorized party. So, for such data owners, the data needs to be encrypted before outsourcing to the third party cloud server. Although the encryption of data ensures its security, it impedes the cloud server's ability to perform search operation. Thus, there is a need for a scheme which can provide a reasonable trade-off between search speed and data privacy.

1.1 Motivation

The domains that we have discussed so far require searching of complete words. But, the problem of searching for occurrence of sub-strings in the words is more important and needs to be solved, as it has direct impact on numerous real life applications. The fact that the number of sub-strings in a string is quadratic in the length of the string, makes the problem difficult to solve efficiently. Some of the motivating scenarios where the solution to this problem is indispensable are:

- Searching for a contiguous pattern in a DNA sequence
- Auto-complete feature of search engine

This problem has only been solved in public key domain. Public key based schemes are not suitable for deployment in cloud scenario as they require multiple rounds of search and also their search and communication complexity is linear in the query or text size. Moreover, public key schemes, though very expressive, require client side computation. In cloud scenario, clients may not be able to perform expensive operations. So far, there has been no scheme in symmetric key domain which can solve this problem. Our work is motivated to solve this problem of searching sub-strings within words in symmetric key domain. Hereafter, we call this problem as string pattern matching.

1.2 Problem Statement

The problem of string pattern matching is defined as searching for a sequence of contiguous characters of length m in a text of length n characters. A more formal definition of string pattern matching is given as: Searching for occurrence of a pattern $P \epsilon [\Sigma]^m$ in a text $T \epsilon [\Sigma]^n$, where Σ is the universe of all characters. String patterns that we focus on are of two types:

Prefix pattern P is a sequence of continuous characters of length m which occurs at the beginning of text T of length n. In other words, $P = T[0, m-1]$.

Sub-string pattern P is a sequence of continuous characters of length m which occurs anywhere within the text T of length n. In other words, $P = T[i, i+m-1]$, where $0 \leq i \leq n-m$.

In the thesis, string pattern matching has a slightly different definition. Outsourced data is represented as a collection of documents, where each document consists of set of keywords, sequence of

characters. String pattern searching on outsourced data is, thus, defined as string pattern matching on keywords of documents and returning documents which are relevant to the string pattern. 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. Thus, documents containing these keywords would be returned as a result.

To describe the problem in the context of cloud computing, initially, the data owner has a set of documents, $\mathbf{D} = \{D_1, D_2, ..., D_M\}$ where each document contains a set of distinct keywords, $D_i(W) = \{w_1, w_2, ..., w_n\}$. Before outsourcing the data to the cloud, the data owner computes an index 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 trapdoor using the string pattern query p and submits it to the cloud server. The server processes the trapdoor on the index I and retrieves all documents, D_i such that, p matches at least one keyword $w \in D_i(W)$ for $1 \leq i \leq M$. There are two major challenges in this process: (a) the index I should not leak any information regarding the keywords, such as size, number, content, etc. and (b) the string query processing technique should be efficient and scalable to large collections of keywords.

1.3 Adversary and Security Model

We adopt the semi-honest adversary model [10] 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 [20]. 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) [16, 50] scheme. To prove security in IND-CKA model, an adversary is allowed to choose two distinct document sets D_0 and D_1, containing nearly equal number of keywords and with some amount of overlapping keywords across the sets. The adversary submits the document sets to an oracle, which randomly constructs a secure indexes I_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. Also, as a symmetric encryption scheme is deterministic, the same pattern will generate the same trapdoor and the cloud server will return the same results [16].

1.4 Limitations of Prior Art

There are numerous state-of-the-art solutions for achieving secure string pattern matching under the secure multiparty computation model [3, 30, 42, 53]. These solutions achieve complex secure pattern matching functionality, but have multi-round communication complexity and involve expensive computations, such as exponentiations and garbled circuit evaluations. Furthermore, the general bandwidth requirement in these protocols is equal to the keywords being examined for the query pattern, which clearly is not suitable for the cloud storage model.

Several symmetric key based searchable encryption techniques [11, 12, 13, 16, 20, 27, 44, 50, 51] 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 [37] 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 solve the privacy preserving string query pattern matching problem considered in this 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.

1.5 Thesis Outline

The thesis begins with a basic implementation of string pattern matching with privacy, which has a linear search complexity. This is further optimized to give a hierarchical tree-based structure which achieves logarithmic search complexity, which forms the basis for the schemes proposed in the thesis. Security is further increased by proposing a polynomial evaluation to hide file IDs in the schemes. The above schemes are combined with relative ranking of sub-string patterns to give the first proposed scheme called PASStree.

Further, it is analysed that search can still be optimized by clustering similar words in tree index, which forms the second proposed scheme called PASStree+.

1.6 Contributions of Thesis

Our key contributions are as follows.

- 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.

- We describe PASStree, a secure searchable index structure, which processes a pattern query in logarithmic time complexity for each possible matching keyword.
- 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.
- We prove the security of our algorithm under the semantic security against adaptive chosen keyword attack IND-CKA.
- 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.

1.7 Organization of Thesis

The remaining sections are organized as follows.

Section 2 lists the related works in detail. Section 3 explains the system model and covers the notations along with the prominent data structures used in the thesis. Section 4 gives the detailed schemes of the thesis. Section 5 gives a thorough security analysis of the schemes. Section 6 gives experimental results and finally, section 7 gives the conclusion of the thesis work.

Chapter 2

Related Work

The prior research in the domain of secure search on outsourced data is predominantly classified into two classes:

2.1 Public Key Encryption (PKE) Schemes

PKE schemes are built upon public key cryptographic algorithms as baseline (e.g., ElGamal, RSA, etc.) and use the principle that two parties will jointly perform some computation and the result will be revealed only to one of the parties. Public key based schemes commonly use methods like homomorphic encryption, oblivious transfer, etc. PKE schemes are further classified as:

2.1.1 Secure Pattern Matching (SPM) schemes

SPM schemes are directly related to our work as they tackle the same problem of pattern matching. First efficient implementation of pattern matching was given by Knuth, Morris and Pratt, called KMP algorithm [32]. It had computation complexity linear in size of text. This was extended to security domain by [53] which had complexity of $O(mn)$ where m and n are pattern size and text size respectively. [53] gave a scenario of genome testing where secure pattern matching is required. After this many works [3, 26, 65, 64, 54, 17, 23, 2] came up with more efficient implementations of pattern matching in genome testing. [3] gave a linear complexity method with constant rounds of interaction between client and server which provided pattern matching in presence of malicious adversaries. Also, their method could hide pattern size. [64] gave a pattern matching method which had lesser bandwidth requirement than [3]. [54] proposed a method which they claimed to provide a stronger notion of security, namely Universally Composable (UC) security. [23] proposed hamming distance based approximate pattern matching in presence of malicious adversaries and gave two separate implementations for hiding pattern length and hiding text length respectively. [17] gave the first scheme which provided both position and size hiding of pattern. Their scheme is secure against Honest but Curious (HbC) adversaries. [26, 65] gave secure similarity computation between documents based on hamming distance in

presence of malicious adversaries. [2] gave many applications of secure pattern matching in genomic testing and gave many Private Set Intersection (PSI) schemes for the various applications. Thus, they showed that PSI schemes are directly related to secure pattern matching scenarios.

2.1.2 Private Set Intersection (PSI) schemes

Various PKE based PSI schemes [9, 48, 18, 4] were proposed which perform pattern matching in scenarios where pattern size and text size are comparable. Scheme of [4] performs approximate string matching between strings of two parties and tells one of the parties whether the similarity is within a threshold or not. It also hides exact distance and location of matching and provides security against HbC model. [18] gives a scheme which performs PSI using oblivious bloom filter intersection and achieves security in malicious model. [9] evaluates hamming distance between inputs of two parties and also achieves security in malicious model.

2.1.3 Deterministic Finite Automata (DFA) Evaluation schemes

Various PKE based DFA Evaluation schemes [42, 59, 60] were also proposed for solving pattern matching problem, and thus are related to our scheme. Wei et al. [59] gave a DFA evaluation scheme in which server has an encrypted file and client has a DFA which they evaluate together and client gets the final state of DFA. Their scheme is secure against malicious server and HbC client. They further enhanced this scheme to give another scheme [60] in which client can verify if server is deviating from the protocol. Mohassel et al. [42] gave a DFA evaluation scheme in which client's computation complexity is only dependent on its input size. They extended it to solve various pattern matching problems.

2.1.4 Secure Two-Party Computation using Garbled Circuits

Other schemes that were proposed in public key domain were [63, 1, 5, 28] which perform two-party computation in constant rounds by solving garbled circuits in a non-interactive way. Though these schemes do not tackle the problem of secure pattern matching, but they are worth mentioning as they are general purpose secure two-party computation schemes which can be used to perform DFA evaluation and for structure encryption. The concept of garbled circuits was first introduced in [63] where it was used to solve secure two-party computation. [5] gave a more formalized definition and security analysis of garbled circuits. [1] extended garbled circuits to arithmetic circuits, and [28] created a special purpose garbled circuit using structure encryption for DFA evaluation and other applications.

2.2 Searchable Symmetric Encryption (SSE) Schemes

All the above PKE schemes use heavy computation methods like additively homomorphic encryption and/or oblivious transfers (OTs). PKE schemes usually require multiple rounds of interaction between two parties and also require high bandwidth. Moreover, they have linear search complexity in terms of pattern size or text size and are thus slower compared to SSE schemes, which usually have sub-linear or logarithmic search complexity. Thus SSE schemes are more practical for cloud scenario.

2.2.1 Exact Keyword Search Schemes

Exact keyword search is the most basic type of search where presence of a keyword in a document is tested for and is most commonly used in the Information Retrieval (IR) community. We note that exact keyword search is a special case of pattern search. [50] was the first work to perform exact keyword search with linear complexity. [20] gave another scheme with security proofs and linear search complexity using bloom filters, but incurred false positives. Their scheme was secure under indistinguishability [21, 8] against chosen keyword attack (IND-CKA). [13] used the same security definitions and achieved linear search complexity without false positives. [16] was the first scheme to achieve sub-linear search complexity and was secure under indistinguishability against adaptive chosen keyword attack (IND-CKA2). [35] proposed a scheme which was secure against a stronger notion of security called universally composable (UC) secure, but it had linear search complexity. Finally, [27] gave a scheme for sub-linear (logarithmic) time exact keyword search which provided adaptive security and could be efficiently parallelized. [39] proposed a scheme for secure keyword search on outsourced personal health records of patients, which can be queried only through a third-party which provides the query trapdoor.

[56, 67, 55, 66] proposed efficient schemes for document ranking and top-k documents retrieval for keyword search. [11] presented a scheme which could handle multi-keyword search in linear time complexity with respect to number of documents. [43, 51, 38, 52, 14] came up with efficient implementations for multi-keyword search supporting ranking. Although our scheme also supports ranking, but the fundamental difference is that the existing schemes rank only documents, while our scheme ranks according to both document relevance and occurrence position of string pattern.

2.2.2 Similarity Based Search Schemes

Fuzzy keyword search tackles the problem of typo in a search query. Though it is not directly related to string pattern search but is an interesting problem worth mentioning. [37] gave an efficient scheme for handling fuzzy keyword search by providing wild-card based and gram based methods. [58] gave another method for tackling fuzzy keyword search by performing k-nearest neighbour (kNN) based query. [57, 36, 33, 34, 24] proposed several schemes for similarity search on outsourced data using

similarity measures like edit distance, etc. [15] came up with a scheme which supports multi-keyword fuzzy search.

We note that these schemes cannot be adopted for solving string pattern matching without exponentially blowing up the index size.

2.2.3 Other Schemes

A more general and powerful concept called Oblivious Random Access Machines (ORAMs) [22] was developed which could be used in both SSE and PKE schemes to provide the most stringent security in a sense that no information is leaked from the index, not even access pattern. The scheme could be adopted to perform many types of search, but it is impractical in a sense that it requires client to have storage space logarithmic to number of documents at the least. A SSE scheme for secure pattern matching was proposed [19] that is orthogonal to our goals in that it deals with search and editing of patterns in strings and our aim is to perform efficient search only. Recently, [44] proposed a scheme consisting of a tree of bloom filters built upon documents, which provides scalable DBMS queries. It is similar to our scheme in structure but differs in storage and search functionality from our scheme and does not support string pattern search.

Chapter 3

System Model and Definitions

3.1 System Model

The cloud scenario (Figure 3.1) consists of 3 parties, namely:

- **Data Owner** is the owner of data to be outsourced. Data owner encrypts data and outsources it to cloud server along with a search index.

- **Cloud Server** is the cloud service provider which stores the outsourced data and performs secure search on the data. Cloud server is considered to be honest-but-curious, i.e., he will perform query and will not deviate from the scheme, but might be interested to infer information from query and search index.

- **Data User** is the client authorized by the data owner to perform query on outsourced data. Data user has the key to decrypt the encrypted data. The key is shared between data user and data owner through secure key sharing protocol which is not part of the thesis.

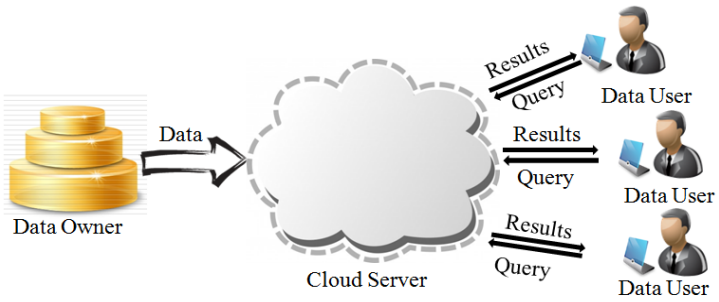


Figure 3.1 System Model

3.2 Design Goals

- **Pattern Search:** The schemes should support string pattern matching on document set and return documents satisfying the pattern matching.
- **Ranking of Results:** Documents returned should be ranked according to the position of occurrence of pattern in the documents and also according to relevance of pattern with respect to documents.
- **Security Goals:** The index should not reveal the documents without performing an authorized search and also should not reveal the similarity of documents based on content. Moreover, query pattern should also not be revealed.
- **Efficiency:** Search should be performed in logarithmic time complexity. There should be no overhead on client side.

3.3 Notations

The notations used in the thesis are:

Σ	Alphabet set (universe of all characters)
$w \in \Sigma^l$	Keyword w of length l characters
$W = \{w_1, w_2, ..., w_N\}$	Dictionary of all N keywords
$\mathbf{D} = \{D_1, D_2, ..., D_M\}$	Document collection **D** is a set of M documents which is encrypted and outsourced
$DL(w) = \{D_1, D_2, ..., D_v\}$	Document listing of w, i.e., list of documents containing w
$EP(x) = (x-r_1)(x-r_2)...(x-r_d) = c_d x^d + c_{d-1} x^{d-1} + ... + c_0$	Encoding polynomial of degree d where r_1, r_2, ..., r_d are the roots and c_0, c_1, ..., c_d are coefficients of the polynomial
$r, c \in G_e$	Root r and coefficient c are elements of a finite field G_e, where e is a large prime
$L = \{w_1, w_2, ..., w_n\}$	List L of n keywords
$\|L\|$	Cardinality of list L

Table 3.1 Notations

3.4 Data Structures

In this section, we describe the various data structures that we use in our work.

3.4.1 Bloom Filter

Bloom filter [6] is a data structure which provides constant time store and lookup operations. It is essentially an array of m bits, all initially set to 0. A bloom filter is accompanied by a set of k hash functions which are used for store and lookup. When an element has to be inserted into bloom filter, its k numeric hashes are calculated with the hash functions and corresponding bit positions in bloom filter are set to 1. For lookup, again k numeric hashes are calculated and corresponding bit positions are checked. If all k bits are set to 1, the element is present in the bloom filter.

Bloom filters can cause false positives, i.e, they can declare that an element is present while it is actually not present. This occurs when multiple bits corresponding to queried element are already set to 1 by other elements present in the bloom filter.

Theorem 1. *Bloom filter of size m with k hash functions has a false positive rate of $(1-e^{-kn/m})^k$ after storing n elements.*

Proof. If 1 bit is set to 1, probability of a bit being set is: $\frac{1}{m}$

$\therefore$ probability of a bit not being set is: $1 - \frac{1}{m}$

If k hash functions are present, then after storing an element, probability of a bit not being set is: $(1 - \frac{1}{m})^k$

After inserting n elements, probability of a bit not being set is: $(1 - \frac{1}{m})^{kn}$

$\therefore$ probability of a bit being set after inserting n elements : $1 - (1 - \frac{1}{m})^{kn}$

For lookup of an element, k bit positions will be checked. Probability of all k bits being set to 1 : $(1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{-kn/m})^k$

Thus, false positive rate is $(1 - e^{-kn/m})^k$ □

Thus, by fixing the number of hash functions (k), the accuracy of bloom filters vary with the ratio of $\frac{m}{n}$, i.e., the number of bits per element stored. Experimental results [7] suggest that for $k = 7$, 10 bits per element inserted give 1% false positive rate (i.e., 99% accuracy). This accuracy is suitable for practical applications and thus $\frac{m}{n}$ value of 10 and $k = 7$ is used in all the experiments of the thesis.

3.4.2 Encoding Polynomial

Encoding polynomial [68] is a polynomial $EP(x)$ of degree d with d roots $r_1, r_2, ..., r_d$ representing the elements stored in the polynomial. The roots are integer values which are bounded by a large prime e.

$$\begin{aligned} EP(x) &= (x - r_1)(x - r_2)...(x - r_d) \\ &= c_d x^d + c_{d-1} x^{d-1} + ... + c_0 \end{aligned} \tag{3.1}$$

Where c_i's are the coefficients of EP for all $i \in [0, d]$ and are bounded by a large prime e. To check for the presence of an element x' in $EP(x)$, $EP(x)$ is solved with $x = x'$ and it evaluates to 0 only if x' is a root of $EP(x)$.

$$EP(x') \begin{cases} = 0 & \text{if } x' \text{ is a root of } EP(x) \\ \neq 0 & \text{if } x' \text{ is not a root of } EP(x) \end{cases} \tag{3.2}$$

Encoding Polynomial EP is stored as a set of coefficients c_i's for all $i \in [0, d]$ which are in the range of $[0, e-1]$. Thus all the roots and coefficients of EP are elements of a finite field G_e [61, 21] and all the operations of EP creation and evaluation are performed *modulo* e, so that the result of the operations always lie within G_e. Storing only the coefficients guarantees that the corresponding roots will not be revealed for a polynomial of sufficiently large degree d, but provides sufficient information for polynomial evaluation for a given root.

Chapter 4

Our Approach

The initial step of data preprocessing is explained before moving to the details of the proposed schemes for string pattern matching. After the preprocessing step, a basic scheme of string pattern matching is explained which preserves the privacy of the data but has a search time linear to the order of number of keywords in the index. This is extended to form a hierarchical tree structure which efficiently supports binary search on the index. Next, a polynomial scheme is proposed for further enhancing the security of the overall scheme by encrypting the Document Identifiers of the document listings.

An efficient scheme which combines the above mentioned techniques and also supports relative ranking of sub-string patterns with the help of an additional structure is then proposed which is called PASStree (Pattern Aware Secure Search tree).

Second scheme that is proposed is the PASStree+ (Pattern Aware Secure Search tree plus) which takes advantage of the clustering concept to further increase the efficiency of search.

4.1 Preprocessing

Give a document collection **D** consisting of M documents (D_1 to D_M), each consisting of a set of keywords from a dictionary W, the preprocessing step consists of extraction of keywords and forming a document listing for each keyword, which consists of all documents in which the keyword occurs. The document listings are sorted in decreasing order of relevance to the keywords. The work uses some of the state-of-the-art techniques of information retrieval and natural language processing domains for performing preprocessing of document(data) collection. The various techniques used for data preprocessing are listed below:

4.1.1 Stop-word removal

In text documents, often there are some words which occur very frequently and are of not much importance. Such words are termed as *stop-words*, and they can be pruned without affecting the efficiency

of the scheme. Pruning of stop-words effectively reduces the size of the index scheme. Some of the examples of stop-words are: *is*, *an*, *of*, *the*, *their*, etc.

The words remaining in the text documents after stop-word removal are called *keywords*. The keywords form an essential part of index scheme and query.

4.1.2 Stemming

As stated by [41], *stemming* is a process of reducing all words with the same root to a common form, usually by stripping each word of its derivational and inflectional suffixes. Stemming not only saves space by reducing the characters in a keyword, but also groups words with same root. For example, different words like *stemmer*, *stemming* and *stemmed* are all reduced to a common word *stem*.

The thesis uses the standard *Porter Stemmer* [46] for all experiments.

4.1.3 Document Listing

Document Listing for a keyword is the set of documents in which the keyword occurs. The documents are sorted in decreasing order of *importance* for the keyword. The importance is measured by the frequency of occurrence of the keyword in the document. But solely calculating the importance based on frequency count can make the measure biased towards the frequently occurring terms, which might not be of much importance.

To solve this, we use the *tf-idf* [62] metric which scales the importance measure. It stands for term frequency-inverse document frequency, and is given as:

$$\begin{aligned} tf - idf(w, D, \mathbf{D}) &= tf(w, D) \times idf(w, \mathbf{D}) \\ &= tf(w, D) \times \log \frac{M}{|\{D \in \mathbf{D} : w \in D\}|} \end{aligned} \tag{4.1}$$

where w is the term/keyword, D is the document and M is the number of documents in the document collection $\mathbf{D}$.

The document listing of all keywords form the inverted index [49], which is a state-of-the-art scheme for efficient search and retrieval in information retrieval domain.

4.1.4 Maximal String Extraction

Maximal Strings [45] are those strings which do not occur as a prefix of any other string in the same document. Since maximal strings subsume all the non-maximal strings, the non-maximal strings can be neglected without any loss of information. Thus the size of the index scheme reduces because only maximal strings are considered as keywords.

If a document D contains two strings *war* and *warship*, then *warship* is the maximal string as *war* occurs as its prefix and thus *warship* subsumes *war*. No information is lost by neglecting *war* as all

sub-strings of *war* occur at the same positions in *warship*, and thus it does not affect the searching of those sub-strings.

It is noted that considering maximal strings is just an additional advantage which reduces the index size of the schemes. The proposed schemes perform efficiently even without considering the maximal strings.

4.2 Basic String Pattern Matching with Privacy

A naive method of searching for a string pattern (sub-string/prefix) is to enumerate all possible sub-strings of a keyword and perform exact pattern match with the obtained sub-strings. Thus the problem of string pattern matching is reduced to equality checking.

Algorithm 1: Naive Pattern Search

```
input  : pattern P, keyword T
output: print if pattern P is found in T
foreach sub-string S of T do
    if P = S then
        print "Pattern Found";
        return the document listing of T;
    end
end
```

For a keyword of length l, there are $\frac{l\times(l+1)}{2}$ combinations of sub-strings. Thus, for pattern matching with N keywords of average length l, the time complexity is $O(N \times \frac{l\times(l+1)}{2})$.

This can be reduced by storing all sub-strings of a keyword in a bloom filter. Thus, each keyword will have a bloom filter, and the problem of string pattern matching is now converted to membership test of bloom filters. The string pattern is searched in each keyword's bloom filter and if it is found in the bloom filter, the pattern is said to be present in the respective keyword. By storing all the sub-strings of a keyword in a bloom filter, we need not enumerate the sub-strings every time while searching. Moreover, the bloom filters are space efficient data structures. For example, for storing a string of 10 bytes, bloom filter will require only 10 bits.

Initially, when the bloom filter B of size m bits is empty, all m bits are set to 0. For storing a string S in B, k hash values of S are computed using k hash functions (Algorithm 2) and the locations in B corresponding to the hash values are set to 1. It is noted that the value of m should be a prime number for better distribution of modular operations [61, 29].

Algorithm 2: Storing in Bloom Filter

input : string S of l bytes, bloom filter B of size m bits, k hash functions H_1, H_2, ..., H_k
output: store string S in bloom filter B
initialize h;
for $i \leftarrow 1$ **to** k **do**
    $h \leftarrow H_i(S)$;
    set $B[h\%m]$ to 1;
end

For searching a string S in B, k hash values of S are computed using k hash functions (Algorithm 3) and if all the locations in B corresponding to the hash values are set to 1, then S is said to be found in B. If even one of the k locations is 0, then S is not stored in B.

Algorithm 3: Searching in Bloom Filter

input : string S of l bytes, bloom filter B of size m bits, k hash functions H_1, H_2, ..., H_k
output: return *true* if S is found in B, else return *false*
initialize h;
for $i \leftarrow 1$ **to** k **do**
    $h \leftarrow H_i(S)$;
    if $B[h\%m] = 0$ **then**
        return *false*;
    end
end
return *true*;

The above algorithms for storing and searching in bloom filters have a security issue that if same string S is inserted into multiple bloom filters, then the same corresponding bits are set to 1 in all the bloom filters as we are using same k hash functions for storing. This leaks some statistical information about the content of the bloom filters. To avoid this, [20] suggested a secure method of storing and searching by performing double hashing using a cryptographic hash (HMAC-SHA1) function, which takes a 160 bit hash key and a variable length message as input and produces a 160 bit hash value. Also, statistical inference can be made on the number of strings stored in a bloom filter, by the number of bits set. To thwart such inferences, some random bits must be set. [20] explains in detail about setting random bits without affecting the false positive rate of bloom filters.

For secure storing (Algorithm 4) in bloom filter B, first an identifier B_{ID} for bloom filter is created, which is unique for each bloom filter. Then, k random hash keys are created as h_1, h_2, ..., h_k. Double hashing is performed for storing string S in B using B_{ID} and k hash values. In first hashing, S is hashed with hash key h_i to produce a hash value v_1. In second hashing, B_{ID} is hashed with v_1 as hash key to produce a hash value v_2. The location in B corresponding to v_2 is set to 1. This process is repeated for each value of $i \in [1, k]$, as depicted in Figure 4.1.

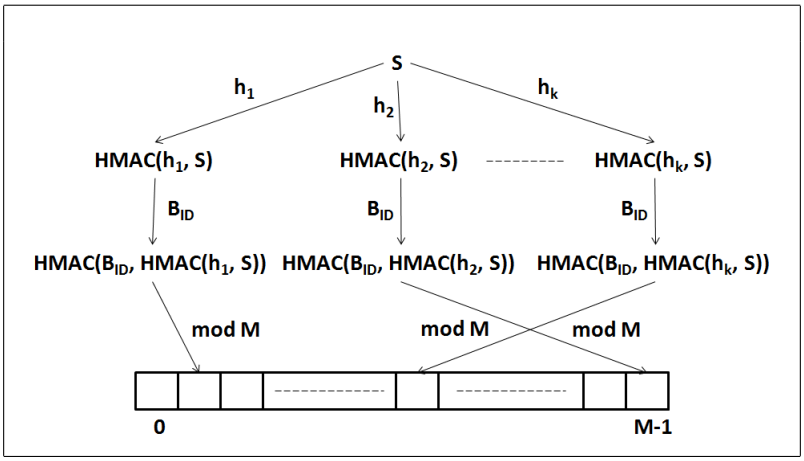


Figure 4.1 Hashing in Bloom Filter

Algorithm 4: Secure Storing in Bloom Filter

input : string S of l bytes, bloom filter B of size m bits, bloom filter ID B_{ID}, k hash keys h_1, h_2, ..., h_k

output: store string S in bloom filter B

initialize v_1, v_2;

for $i \leftarrow 1$ **to** k **do**

  $v_1 \leftarrow$HMAC-SHA1(h_i, S);

  $v_2 \leftarrow$HMAC-SHA1(v_1, B_{ID});

  set $B[v_2 \% m]$ to 1;

end

set some random bits to 1;

For secure searching (Algorithm 5) in B, same process is performed for double hashing to get the corresponding locations in B.

Algorithm 5: Secure Searching in Bloom Filter

input : string S of l bytes, bloom filter B of size m bits, bloom filter ID B_{ID}, k hash keys h_1, h_2, ..., h_k

output: return *true* if S is found in B, else return *false*

initialize v_1, v_2;

for $i \leftarrow 1$ **to** k **do**

  $v_1 \leftarrow$HMAC-SHA1(h_i, S);

  $v_2 \leftarrow$HMAC-SHA1(v_1, B_{ID});

  if $B[v_2 \% m] = 0$ **then**

    return *false*;

  end

end

return *true*;

Thus, bloom filters provide constant time storage and lookup operations. They are extended to store all the sub-strings of the keywords. Thus the basic scheme for string pattern search has N bloom filters for N keywords. The Algorithm 6 for storing in the basic scheme is given below:

Algorithm 6: Basic Scheme for Storing

input : N keywords w_1, w_2, ..., w_N of l bytes

output: store sub-strings of keywords in their respective bloom filters

initialize B_1, B_2, ..., B_N of size m bits to 0;

for $i \leftarrow 1$ **to** N **do**

  foreach *sub-string* S *of* w_i **do**

    store S in B_i using Algorithm 4;

  end

end

The Algorithm 7 for searching in the basic scheme is given below:

Algorithm 7: Basic Scheme for Searching

input : N bloom filters B_1, B_2, ..., B_N of size m bits, pattern P

output: print all the keywords in which the pattern is found and return their document listings

for $i \leftarrow 1$ **to** N **do**

  search P in B_i using Algorithm 5;

  if P *is found in* B_i **then**

    print "Pattern P found in w_i";

    return the document listing of w_i;

  end

end

Thus, the time complexity of pattern matching with N keywords is reduced to $O(N)$, which is linear only in the number of keywords.

4.3 Using Hierarchical Tree Structure

The basic scheme cannot achieve a lesser bound on time complexity than $O(N)$, which is linear to the number of keywords in the document collection. We extend the basic scheme to a hierarchical tree structure which achieves logarithmic time complexity but at the cost of increase in space complexity from $O(N)$ to $O(NlogN)$.

We form a highly balanced binary tree structure in which each leaf has a bloom filter corresponding to a keyword in the dataset. The tree is constructed in such a way that at each level, the parent node's bloom filter stores all sub-strings stored in child nodes' bloom filters. Thus, the root node of the tree structure stores all the sub-strings occurring in the document collection.

Theorem 2. *(Logarithmic Complexity): For a given query string p, the PASStree search algorithm finds all matching leaf nodes in a complexity of $O(logN)$ where N is the total number of leaf nodes in the PASStree.*

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 *left* and *right* nodes. 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 $2logN$ 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 $2logN$, the aggregated complexity of searching for pattern p is $2ElogN$. This result proves that the PASStree achieves logarithmic complexity in worst-case and achieves sub-linear complexity if a pattern is not found in many keywords. □

The hierarchical tree of bloom filters is built in a top-down fashion (Algorithm 8). At the root node, all the sub-strings of all keywords are stored in the bloom filter. At each level, keywords are split into two approximately equal partitions, one for each child node. Thus, in a similar way, the child nodes are formed at each level of the tree by recursively partitioning the keywords. The leaf nodes represent exactly one keyword and their bloom filter contains sub-strings of that keyword.

Algorithm 8: Building the Hierarchical Tree

input : root node *root*, list L of n keywords w_1, w_2, ..., w_n of l bytes each
output: Create the hierarchical tree of bloom filters
initialize *root*;
initialize bloom filter B of *root*;
foreach w *in* L **do**
    foreach *sub-string* S *of* w **do**
        store S in B using Algorithm 4;
    end
end
if $|L|$ = *1* **then**
    add document listing of keyword of L;
    return;
end
split L into two mutually exclusive lists L_1 and L_2 of size $|L|/2$ each;
recursively call for $root \rightarrow left$ and $root \rightarrow right$ with L_1 and L_2 resp.;

For string pattern searching (Algorithm 9), pattern P is searched in root node's bloom filter and if match is found, search is proceeded towards child nodes. In this way, we recursively search in child nodes if match is found, till we reach a leaf node. If match is found in the leaf node, the document listing of that node is returned. In this fashion, all leaf nodes which contain the pattern are searched and corresponding document listings are returned in a single round.

Algorithm 9: Searching in Hierarchical Tree

input : root node *root*, pattern P
output: return the document listings of the keywords containing the pattern
search P in bloom filter B of *root* using Algorithm 5;
if P *found in* B **then**
    if *root is leaf node* **then**
        return document listing of *root*;
    else
        recursively call for $root \rightarrow left$ and $root \rightarrow right$ with P;
    end
end

Since there are $log(N)$ levels in the tree and each keyword is stored in exactly one node at each level, the overall space complexity is $O(NlogN)$. Search complexity is $O(logN)$. Thus, with this scheme, we obtain logarithmic search complexity with slight increase in space complexity.

4.4 Polynomial Scheme for hiding Document Identifiers

In the hierarchical tree scheme, a passive adversary can still examine the leaf nodes and attempt to correlate the number of keywords shared between two documents. While this leakage is allowed [16] eventually, i.e., after the cloud server has executed several queries, this cannot be allowed by direct examination. Specifically, for an index to be secure in the IND-CKA model, the privacy leakage cannot be allowed for keywords that have not yet matched any pattern query. To prevent this leakage, we encrypt each document listing, and then, allow the cloud server to decrypt the corresponding document listing only when the user submits a valid pattern query trapdoor.

Document List Encryption. To provide security to the document listing, we use the onion encryption [16]. The document listing is encrypted in a chained manner, such that, decrypting the first node gives the decryption key of the second node and so on. Given a list of ordered document identifiers, $DL = \{D_1, D_2, ..., D_v\}$, the data owner generates a set of encryption keys: $\{K_1, K_2, ..., K_v\}$, for each of the identifier respectively. The first identifier, D_1 is encrypted using a symmetric key encryption algorithm $Enc(.)$ as follows: $Enc(K_1, D_1|| < K_2, Addr(D_2) >)$ where $Addr(D_2)$ is a memory location[1]that contains the value: $Enc(K_2, D_2|| < K_3, Addr(D_3) >)$. To parse this list, the cloud server needs to first decrypt the first node using K_1 and parse the information into: D_1, K_2 and uses K_2 to decrypt the node stored in the location $Addr(D_2)$ and terminating the process when the final address points to a null location. The Algorithm 10 for document list encryption is given below:

Algorithm 10: Document List Encryption

input : Document Listing $DL = \{D_1, D_2, ..., D_v\}$
output: Encrypted Document Listing
generate random keys $K_1, K_2, ..., K_{v+1}$;
for $i \leftarrow 1$ *to* $v - 1$ **do**
  $D_i \leftarrow Enc(K_i, D_i|| < K_{i+1}, Addr(D_{i+1}) >)$;
end
$D_v \leftarrow Enc(K_v, D_v|| < K_{v+1}, NULL >)$;
return encrypted DL;

The Algorithm 11 for document list decryption is give below, which uses a symmetric key decryption algorithm $Dec(.)$. Here, $addr$ is the address which stores the next encrypted document identifier in the document listing.

[1]It is to be noted that on cloud server, the address can be path to the location in hard drive or other storage space, whereas if the index is loaded in the main memory, address is the memory location containing the data.

Algorithm 11: Document List Decryption

input : First encrypted document identifier D_1 of the document listing DL, First decryption key K_1

output: Decrypted Document Listing

$D_1, K_2, addr \leftarrow Dec(K_1, D_1)$;

set i to 2;

while $addr \neq NULL$ **do**

  $D_i, K_{i+1}, addr \leftarrow Dec(K_i$,Value at $addr)$;

  increment i by 1;

end

return decrypted DL;

Access Encoding. Now, the challenge is that, as a leaf node contains several patterns, the cloud server should be able to decrypt the corresponding encrypted list for any matching pattern queries. Specifically, the cloud server only needs access to the decryption key of the first node in the encrypted list and therefore, we describe an approach to achieve this. Given a keyword w, the set of patterns is given by, $S(w) = \{p_1, p_2, ..., p_d\}$. We use the scheme from [68], to encode these patterns in such a way that, only a valid pattern from this list will reveal the first decryption key of the encrypted list.

We consider a finite field G_e defined over a large prime number e and a one-way secure hash function: $F : \{0,1\}^* \rightarrow \{0,1\}^e$. For a given leaf node, the data owner generates an access polynomial, defined over the field G_e, as follows:

$$A(x) = \Pi_{i=1}^{i=d}(x - F(T_i, T_{p_i})) \tag{4.2}$$

where the operation $F(x, y)$ is defined as as $HMAC(x||y)\%e$. The value T_i is an m-bit binary string induced by a valid trapdoor T_{p_i}, corresponding to the pattern query p_i, as follows. Each T_{p_i} maps into k distinct locations into the leaf Bloom filter, which is a bit-array of size m. We generate T_i by treating all other locations of the Bloom filter array as zero except the k distinct locations to which T_{p_i} is hashed and copy the resulting m-bit string into T_i. Next, the data owner generates r random values over G_e and computes the polynomial:

$$R(x) = \Pi_{j=1}^{j=r}(x - F(R_j, z)) + C_r \tag{4.3}$$

where z, C_r are random constants that are kept secret by the data owner. R_j is a random m-bit binary string. The purpose of $R(x)$ is to hide the number of real sub-strings, p_i in the keyword, as this information might leak the size of the encrypted keyword. We note that the value of r can be fixed as $r = (d' - d)$ for each keyword, where d' is a fixed constant which defines the maximum number of possible sub-strings and d is the number of sub-strings of the keyword. Finally, the data owner encodes the decryption key K_1 as follows:

$$EP(x) = A(x).R(x) + K_1 \tag{4.4}$$

where $EP(x)$ stands for Encoding Polynomial, and its degree is d'. The intuition of this encoding is that, for a matching pattern, p_i , a corresponding matching T_i is generated from the Bloom filter, and the cloud server evaluates $EP(x)$ at $x = F(T_{p_i}, T_i)$. Since each valid pattern is a valid root of $EP(x)$, the first term of $EP(x)$ reduces to zero and reveals K_1. Finally, as this evaluation works only if T_{p_i} is a valid trapdoor, this approach protects the document listing privacy of keywords, which are not matched against a pattern query.

Algorithm 12 for creating a polynomial for a word w is given below. It uses symmetric key encryption algorithm $Enc(.)$, like AES, for creating trapdoor T_{p_i} for p_i.

Algorithm 12: Create Polynomial

input : Set of sub-strings of keyword $w = \{p_1, p_2, ..., p_d\}$
output: Encoding Polynomial EP for w
initialize C_r and z with random value;
foreach *pattern* p_i *of* w **do**
  $T_{p_i} \leftarrow Enc(p_i)$;
  set T_i by calling Algorithm 2 with T_{p_i};
end
$A(x) \leftarrow \Pi_{i=1}^{d}(x - F(T_i, T_{p_i}))$;
$r \leftarrow d' - d$;
for $j \leftarrow 1$ *to* r **do**
  initialize R_j with random m bits;
end
$R(x) \leftarrow \Pi_{j=1}^{r}(x - F(R_j, z)) + C_r$;
$EP(x) \leftarrow A(x).R(x) + K_1$;

Algorithm 13 for evaluating the polynomial for query pattern p is given below:

Algorithm 13: Evaluate Polynomial

input : query pattern p
output: Key K_1 for decrypting Document Listing
$T_p \leftarrow Enc(p)$;
set T by calling Algorithm 2 with T_p;
$K_1 \leftarrow EP(F(T, T_p))$;

4.5 PASStree

Pattern Aware Secure Search tree (PASStree), combines the hierarchical tree structure and the polynomial scheme and can also identify the relative position of occurrence of pattern in keywords by taking leverage of an additional bloom filter at each node of the tree, called the *Sub-string Prefix Bloom Filter* (SP). Using this property of *identifying* the position of occurrence of pattern in keywords, PASStree

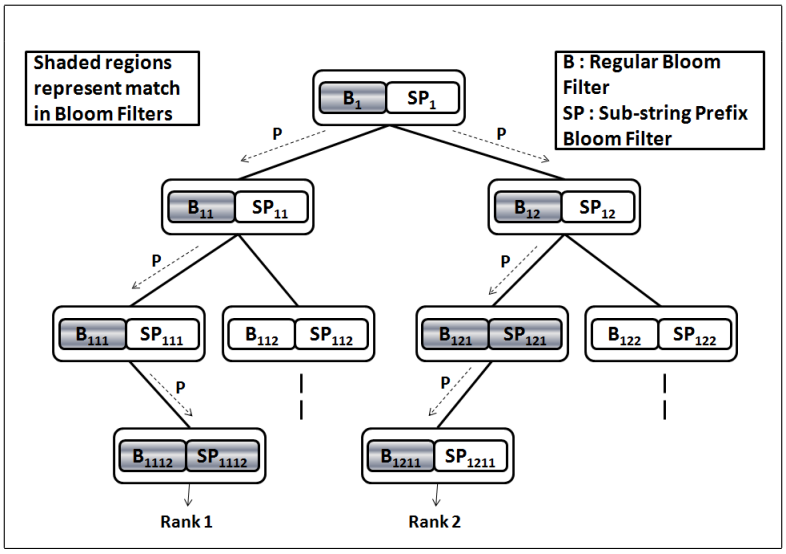


Figure 4.2 Ranking in PASStree

can rank the keywords, and thus the documents in turn. This way PASStree can return the documents in the order of relevance to the query pattern.

For example, if a query pattern *ship* appears in two keywords *warship* and *shipment*, then *shipment* is ranked higher than *warship*, since *ship* occurs at beginning position in *shipment* and at fourth position in *warship*. Hence, *shipment*'s document listing will be returned first and then *warship*'s document listing will be returned as a result of the query execution.

SP bloom filter of a node at a distance h from the leaf node stores all the prefixes of the sub-string starting at h^{th} position of a keyword. For example, for a keyword *warship*, leaf node's SP bloom filter will store $\{$*warship, warshi, warsh, wars, war, wa, w*$\}$. SP bloom filter of its parent node will store $\{$*arship, arshi, arsh, ars, ar, a*$\}$, and so on. At root node, all the remaining sub-strings will be stored in its SP bloom filter. In this way, all the prefix matches will occur in leaf nodes, and we can differentiate the matching keywords based on the occurrence position of the pattern. Figure 4.2 depicts the general structure of the PASStree and also depicts how a query pattern P is searched in PASStree and the relative ranking of leaf nodes according to the pattern P.

The Algorithm 14 for creating a PASStree is given below:

Algorithm 14: Creating PASStree

input : root node *root*, list L of n keywords w_1, w_2, ..., w_n of l bytes each
output: Create the PASStree
initialize *root*;
initialize bloom filter B and sub-string prefix bloom filter SP of *root*;
$u \leftarrow$ depth of leaf node - depth of current node;
foreach w *in* L **do**
    foreach *sub-string* S *of* w **do**
        store S in B using Algorithm 4;
    end
    foreach *sub-string* S *of* w *starting at position* u **do**
        store S in SP using Algorithm 4;
    end
end
if $|L|$ = *1* **then**
    create EP using Algorithm 12;
    add document listing of keyword of L;
    encrypt document listing of keyword of L using Algorithm 10;
    return;
end
split L into two mutually exclusive lists L_1 and L_2 of size $|L|/2$ each;
recursively call for $root \rightarrow left$ and $root \rightarrow right$ with L_1 and L_2 resp.;

The Algorithm 15 for searching in a PASStree is given below:

Algorithm 15: Searching in PASStree

input : root node *root*, pattern P
output: return the document listings of the keywords containing the pattern
search P in bloom filter B of *root* using Algorithm 5;
search P in sub-string prefix bloom filter SP of *root* using Algorithm 5;
if P *found in* SP **then**
    $u \leftarrow$ depth of leaf node - depth of current node;
end
if P *found in* B **then**
    if *root is leaf node* **then**
        evaluate EP using Algorithm 13 to reveal decryption key;
        decrypt document listing of *root* using Algorithm 11;
        return document listing of *root* with rank u;
    else
        recursively call for $root \rightarrow left$ and $root \rightarrow right$ with P;
    end
end

4.6 PASStree+

Though PASStree fulfils all the requirements of string pattern matching, we observe that its efficiency can still be improved. Although, the theoretical bound on search complexity is $O(ElogN)$, where E is the number of keywords matching the string pattern and N is the number of keywords in the dictionary, but in practice, the number of comparisons in intermediate nodes of the tree can be reduced. For example, if two words matching the pattern are located far apart in the tree, we would have to parse through all the intermediate nodes in their paths from root to the matching leaf nodes. But if they are located closed together at leaf level, they will share a longer common path from root to their *lowest common ancestor* [2]and thus, number of intermediate nodes comparison is reduced.

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.

[2]In graph theory and computer science, the lowest common ancestor (LCA) of two nodes u and v in a tree or directed acyclic graph (DAG) is the lowest (i.e. deepest) node that has both u and v as descendants

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 improve search efficiency is to arrange two or more keywords in the same partition by measuring the number of patterns they share in common.

We quantify this metric, using the Jaccard similarity coefficient technique [25], as follows:

$$SS_c = \frac{|S(w_1) \cap S(w_2)|}{|S(w_1) \cup S(w_2)|} \tag{4.5}$$

where $S(w_1)$ is the set of sub-strings of w_1 and 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, we observe that, it would be desirable to group together common patterns that occur within the same document, as it leads 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 more 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. These two metrics are computed using Jaccard similarity technique as follows. First,

$$PS_c = \frac{|L(w_1 \rightarrow w_2)| + |L(w_2 \rightarrow w_1)|}{|L(w_1 \cap \not{w_2})| \cup |L(w_2 \cap \not{w_1})| \cup |L(w_1 \cap w_2)|} \tag{4.6}$$

where $L(w_1 \rightarrow w_2)$ indicates the list of documents in which w_1 and w_2 occur as a phrase; $|L(w_1 \cap \not{w_2})|$ is the number of documents containing only w_1 but not w_2 and so on. Second,

$$DS_c = \frac{|L(w_1 \cap w_2)|}{|L(w_1 \cap \not{w_2})| \cup |L(w_2 \cap \not{w_1})| \cup |L(w_1 \cap w_2)|} \tag{4.7}$$

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:

$$S_c(w_1, w_2) = SS_c + PS_c + DS_c \tag{4.8}$$

It is to be noted that the similarity metric does not reveal the content similarity of the keyword pairs, which is evident from Figure 4.3. As the plot depicts. the similarity score S_c is irrelevant of the content similarity of keyword pairs, and thus, it does not violate the privacy.

Partitioning Using Clustering. We use CLARA [31], a well known clustering algorithm, to partition a keyword set based on the keyword pair similarity. CLARA clusters the keywords around a

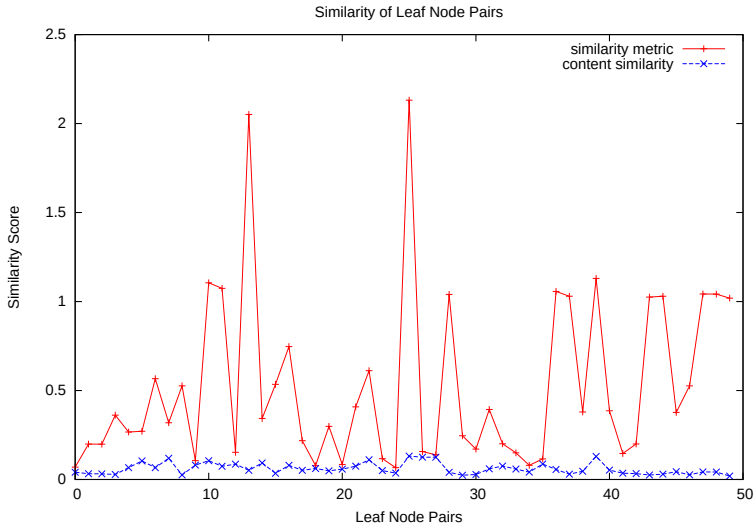


Figure 4.3 Similarity Metric VS Content Similarity

representative keyword, called a medoid, MED, such that all the keywords share a high Sc 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, [31] 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 [31], to get the best possible medoids, we perform 5 iterations as follows.

Medoid Selection and Clustering. Initially, CLARA chooses a set of $L << N$ random keywords from the data set. As per the specification of CLARA, we choose $L = 44$, if $L << N$, and $L = N$, if $N < 44$. Out of these L points: the algorithm chooses two random keywords to act as medoids and clusters the remaining $N - L + 2$ keywords around these two medoids, where the clustering approach is straightforward: a keyword is placed in the cluster of medoid with which its S_c value is the highest. Post clustering, CLARA computes the quality of each of the two clusters, by adding up all the Sc values of the keywords with respect to the corresponding medoids. Next, the algorithm replaces one medoid from another keyword chosen from the $L-2$ keywords and repeats the process. The pair of medoids resulting in the best quality clusters are retained from the L keywords. Repeating this across five iterations, the pair of medoids resulting in the best quality clusters are chosen to partition the keyword set into two clusters. Algorithm 16 for clustering is given below:

Algorithm 16: Clustering using CLARA

input : set of N keywords $W = \{w_1, w_2, ..., w_n\}$
output: partitions P_1 and P_2
if $N < 44$ **then**
  $L = N$;
else
  $L = 44$;
end
for *5 iterations* **do**
  initialize P_1 and P_2;
  select L random keywords from W and store in S;
  select 2 random keywords from S and set as MED_1 and MED_2;
  foreach *keyword* w *in* W **do**
    if $S_c(w, MED_1) > S_c(w, MED_2)$ **then**
      Put w in P_1;
    else
      Put w in P_2;
    end
  end
  calculate $Sim \leftarrow \Sigma_{i=1}^{2} \Sigma_{w \in P_i} S_c(w, MED_i)$;
  foreach *keyword* w *in* S **do**
    replace w with MED_1 or MED_2 and recalculate Sim;
    if Sim *increases* **then**
      accept the replacement;
    end
  end
end
select medoids MED_1 and MED_2 for which Sim is maximum;
perform clustering with MED_1 and MED_2 to get P_1 and P_2;

Balancing the Clusters. The two partitions generated out of the clustering algorithm may not satisfy the conditions of the partitioning problem, and therefore, we need to balance the partitions. Let P_1 and P_2 be the two partitions, where $|P_1| < |P_2|$, and let MED_1 and MED_2 be the two respective medoids. For each $w \in P_2$, we compute $S_c(w, MED_1)$ and sort these values in a descending order. Now, we pick the first keyword of the sorted list and move it to MED_1, and repeat this process, until the partitions satisfy the constraint $||P_1| - |P_2|| \leq 1$. Note that, this process ensures [31] that the quality of the resulting clusters is still the best possible within the chosen medoid keyword samples. Algorithm 17 for balancing the clusters is given below:

Algorithm 17: Balancing Clusters

input : partitions P_1 and P_2, where $|P_1| > |P_2|$
output: balanced partitions P_1 and P_2
initialize L;
foreach *keyword* w *in* P_1 **do**
    calculate $S_c(w, MED_2)$;
end
sort all w's in decreasing order of S_c and store in L;
while $|P_1| - |P_2| > 1$ **do**
    select first word w from L and transfer it from P_1 to P_2;
    remove w from L;
end

The algorithm for creating PASStree+ is similar to that of PASStree, but with an addition of clustering at each node. It is as follows (Algorithm 18):

Algorithm 18: Creating PASStree+

input : root node *root*, list L of n keywords w_1, w_2, ..., w_n of l bytes each
output: Create the PASStree+
initialize *root*;
initialize bloom filter B and sub-string prefix bloom filter SP of *root*;
$u \leftarrow$ depth of leaf node - depth of current node;
foreach w *in* L **do**
    foreach *sub-string* S *of* w **do**
        store S in B using Algorithm 4;
    end
    foreach *sub-string* S *of* w *starting at position* u **do**
        store S in SP using Algorithm 4;
    end
end
if $|L|$ = *1* **then**
    create EP using Algorithm 12;
    add document listing of keyword of L;
    encrypt document listing of keyword of L using Algorithm 10;
    return;
end
partition L into L_1 and L_2 by calling Algorithm 16;
balance L_1 and L_2 by calling Algorithm 17;
recursively call for $root \rightarrow left$ and $root \rightarrow right$ with L_1 and L_2 resp.;

The algorithm for searching in PASStree+ is same as that of PASStree.

4.7 Comparison Of Schemes

We give a comparison of the various proposed schemes in the work and focus that the two schemes PASStree and PASStree+ are most efficient for solving the problem of string pattern matching on outsourced data.

Scheme	String Pattern Matching	Relative Ranking	Security of Document Listing	Space Complexity	Search Complexity
Naive	Yes	No	No	$O(N)$	$O(N \cdot \frac{l \cdot (l+1)}{2})$
Basic	Yes	No	No	$O(N)$	$O(N)$
Hierarchical Tree	Yes	No	No	$O(NlogN)$	$O(ElogN)$
PASStree	Yes	Yes	Yes	$O(NlogN)$	$O(ElogN)$
PASStree+	Yes	Yes	Yes	$O(NlogN)$	$O(ElogN)$

Table 4.1 Comparison of Proposed Schemes
[3]

[3] N is the number of keywords in the data set, E is the number of keywords which match the query pattern and l is the length of the string

Chapter 5

Security Analysis

We analyse the security of the PASStree structure in the semantic security against adaptive chosen keyword attack, IND-CKA, [20] security model against non-adaptive adversaries [16], where a non-adaptive adversary has a snapshot of the communication history of a user with the cloud server. For an index to be IND-CKA secure, a probabilistic polynomial time adversary A should not be able to distinguish [21] between two indexes built over the same size data sets, S_0, S_1, with non-negligible probability, i.e., the advantage of A should be higher than 1/2. The adversary is allowed to see the query trapdoors, the leaf nodes matched, the document identifiers retrieved, and the size of the individual documents. We present arguments about the IND-CKA security of the main structures comprising the PASStree: the Bloom filters, regular and the sub-string prefix (SP), storing the keywords and the encoding polynomial $EP(x)$, used to protect the list of documents.

5.1 Security of Bloom Filters

In the IND-CKA model, the Bloom filters are secure if: (a) the presence of common elements across two Bloom filters cannot be inferred by direct examination of the Bloom filter bits, and (b) the sizes of the Bloom filters do not leak the number of elements stored. Now, to store the keywords in the Bloom filters we use secure pseudo-random functions, $H : \{0,1\}^s \times \{0,1\}^t \rightarrow \{0,1\}^u$ with k symmetric secret keys, $h_1, h_2, ..., h_k \in \{0,1\}^s$ where s is the security parameter of the system. A given Bloom filter in the PASStree can be viewed as a smaller index storing a sub-set of the patterns and therefore, the security of all the Bloom filters can be equivocated [16, 20]. Now, the input to the Bloom filter is a pseudo-random value, $H(h_i, p)$, which is combined with the unique Bloom filter identifier BF_{id} to store the value into a Bloom filter. Therefore, given that the Bloom filter identifiers are distinct and random, the same pattern is stored in different locations across different Bloom filters. As the PASStree nodes at the same distance from the root node store almost the same number of patterns, our PASStree construction algorithm ensures that the size of these Bloom filters is same and that the number of bits are made almost equal by the blinding process, which add an estimated number of random bits to each Bloom filter. During PASStree construction, the blinding process estimates the number of bits required

to normalize the Bloom filters and adds these random bits to each Bloom filter and prevents inference of the number of elements contained in the Bloom filter.

5.2 Security of Encoding Polynomials

The encoding polynomial, $EP(x) = A(x).R(x) + K_1$, is considered secure if: (a) the value of K_1 cannot be compromised unconditionally, and (b) the size of the matching keyword is not leaked. By design, K_1 is information theoretic secure [47] against passive attackers and is revealed to the cloud server, if and only if, a secure trapdoor is available. Next, to prevent the leakage of the size of the keyword, the random polynomial $R(x)$ provides sufficient padding against such inference. However, since K_1 is revealed after a valid trapdoor is searched, the cloud server might recover the polynomial as follows: $A(x).R(x) = EP(x) - K_1$, and then, use factorization techniques to compute the roots of $A(x).R(x)$. But the challenge to the adversary is to differentiate between the coefficients generated by valid trapdoors, i.e. coefficients of $A(x)$ and, the random coefficients, i.e. coefficients of $R(x)$. Towards this, we note that, as the same trapdoor T_r maps in different Bloom filter locations across different leaf nodes, the coefficients induced by T_r , for $A(x)$, are indistinguishable, from those of $R(x)$, to a passive adversary. Without the knowledge of secret keys, the adversarial challenge [40] is equivalent to the advantage of probabilistic polynomial time adversary to differentiate between the output of a pseudo-random function and a random function. Finally, since the security arguments of the Bloom filters and the encoding polynomial are independent, the PASStree structure is IND-CKA secure against adaptive chosen keyword attacks by non-adaptive adversaries.

5.3 Security Game

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 [40]. We have used HMAC-SHA1 for our scheme as the pseudo-random function H and AES as the encryption algorithm Enc for the documents. From [16, 40], 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.

5.3.1 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 is encrypted. 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 $D = \{D_1, D_2, ..., D_n\}$ denote the set of document identifiers where D_i denotes the i^{th} document. The history H_q is defined as $H_q = \{D, p_1, p_2, ..., p_q\}$, where the set 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 T, the secure index I for D and the set of the encrypted documents, $Enc_K(D) = \{Enc_K(D_1), Enc_K(D_2), ..., Enc_K(D_n)\}$, corresponding to the returned document identifiers. Formally, $A_v(H_q) = \{T, I, Enc_K(D)\}$.

Result Trace. This is defined as the *access* and *search* patterns observed by the adversary after T is processed on the encrypted index I. The access pattern is the set of matching document identifiers, $M(T) = \{m(T_{p_1}), m(T_{p_2}), ..., m(T_{p_q})\}$ 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 3. *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 $S = S_0, S_q$ simulating an adversary's view with non-negligible probability. We denote the simulated index as I^*, the simulated encrypted documents as, $Enc_K(D^*)$ and the trapdoors as 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 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: $A_v^*(H_0) = \{Enc_K(D^*), I^*, T^*\}$, 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 I, i.e., this simulated index satisfies all the structural properties of a real PASStree index. Now, given that HMAC-SHA1 is pseudo-random and the probability of trapdoor distribution is uniform, the index I^* is indistinguishable by any probabilistic polynomial time adversary.

Step 3. Simulator StateS_q For H_q where $q \geq 1$, we denote the simulator state by Sq. The simulator constructs the adversary view as follows: $A_v^*(H_q) = \{Enc_K(D^*), I^*, T^*, T_q\}$ 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 $q < |D|$, the simulator generates $1 \leq i \leq |D| - q + 1$ 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 sub-set 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 ST_q to remember the association between the trapdoors and the matching document identifiers. The simulator outputs: $\{Enc_K(D^*), I^*, T^*, T_q\}$. 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. □

Chapter 6

Experimental Results

We perform thorough experimental analysis of the proposed schemes PASStree and PASStree+ on two real-world data sets *Wikipedia Corpus* and *Enron Email Dataset*.

6.1 Experimental Setup

The system configuration on which the experimentation was performed are:

- C++ programming language
- POLARSSL package for HMAC-SHA1 and AES utilities
- Linux Ubuntu 12.04 64-bit OS
- Intel® Core^{TM} i3-2120 CPU @3.30GHz $\times$ 4
- 4 GB RAM
- 500 GB Hard Disk Space

All the experiments are performed by forming the index structure on the document collections multiple times (5 times) and the results shown are average of the multiple runs. This is done to minimize the errors of any individual run and thus the results are reliable.

6.2 Enron Email Dataset

Enron dataset consists of 0.6 million emails of employees of Enron, out of which we choose 10000 emails as a dataset. Each email consists of *From Address*, *To Address* and *Body of the message*. We extract all the keywords from all the fields and perform query on *Sender-Receiver*, which is a *phrase* query consisting of multiple keywords depicting sender and receiver names. Thus, with *Sender-Receiver* queries, we extract all emails which are from a particular sender to particular receiver(s).

For *Sender-Receiver* queries, the individual keywords from sender and receiver are extracted. Next, the query is performed on individual keywords and the results are accumulated. Some examples of *Sender-Receiver* queries are:

- *Example 1*

  From: Grant Colleean

  To: Market Status

  Thus the query is converted to a set of 4 keywords.

  Query: "Grant Colleean Market Status"

- *Example 2*

  From: Ross Mesquita

  To: Harry Arora, Suresh Raghavan

  Thus the query is converted to a set of 6 keywords.

  Query: "Ross Mesquita Harry Arora Suresh Raghavan"

- *Example 3*

  From: Tim Belden

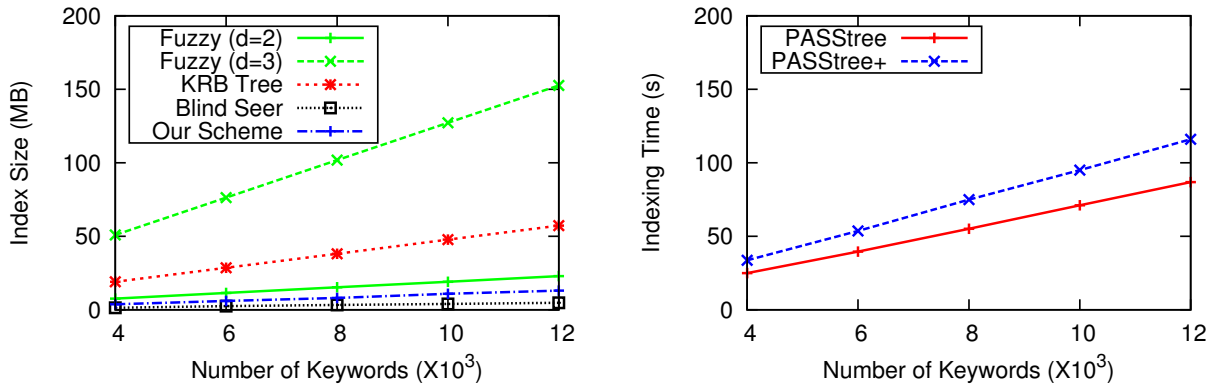
  To: Mike Suerzbin, Robert Badeer, Sean Crandall, Tim Belden, Jeff Richter, John Forney, Matt Motley, Tom Alonso, Mark Fischer

  Thus the query is converted to a set of 20 keywords.

  Query: "Tim Belden Mike Suerzbin Robert Badeer Sean Crandall Tim Belden Jeff Richter John Forney Matt Motley Tom Alonso Mark Fischer"

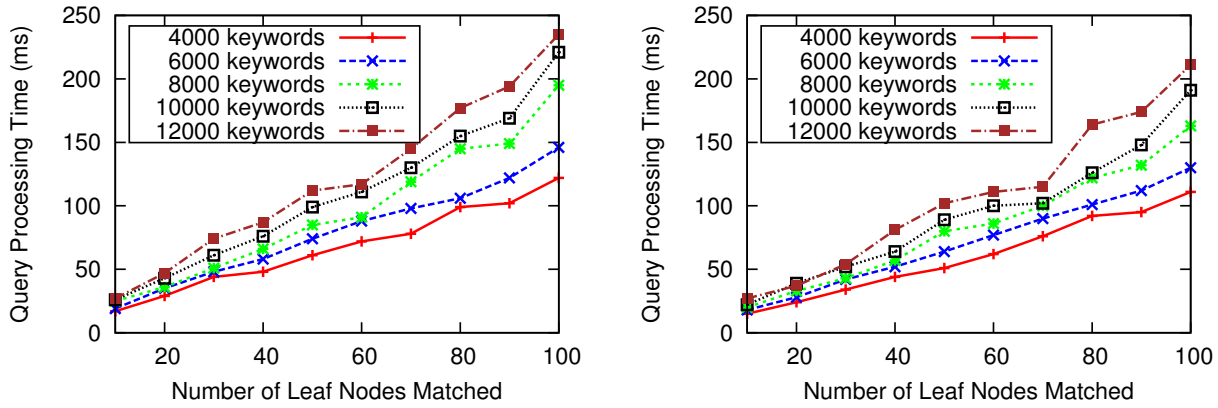
Figure 6.1(a) depicts the index size comparison of our schemes (PASStree and PASStree+) with the state-of-art practical schemes [37, 44, 27] existing in literature. The plot depicts that our scheme is memory efficient and can be used for practical scenarios like cloud. Figure 6.1(b) depicts that the index construction time is linear to the number of keywords, which is reasonable. Also, PASStree+ takes more time for indexing compared to PASStree because of extra overhead of clustering.

Figure 6.2(a) and Figure 6.2(b) depict the query processing time of PASStree and PASStree+ respectively, where each individual line represents the query processing time for index built on different number of keywords. x-axis depicts the number of keywords in the query. The plots depict that PASStree+ fares better than PASStree in query processing, owing to the clustering of similar keywords. It is also noted that the querying takes few milliseconds for different query sizes, supporting the practicality of the schemes.



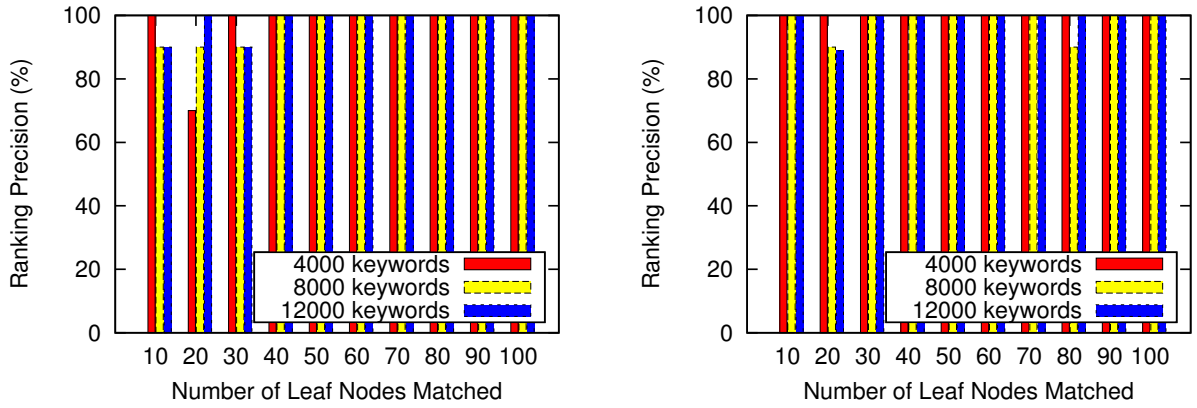
(a) Index Size Comparison (b) Index Construction Time

Figure 6.1 Index construction on Enron Dataset



(a) PASStree (b) PASStree+

Figure 6.2 Query Time on Enron Dataset



(a) PASStree (b) PASStree+

Figure 6.3 Ranking Precision on Enron Dataset

Figure 6.3(a) and Figure 6.3(b) depict the ranking precision of PASStree and PASStree+ respectively, where each individual bar represents the ranking precision for index built on different number of keywords. x-axis depicts the number of keywords in the query. The *ranking precision* [1]is calculated for top ranked documents which are returned. On an average, the ranking precision is 90% - 100%.

6.3 Wikipedia Corpus

Wikipedia corpus consists of 10 million plus web pages, out of which we choose 10000 web pages as a dataset. Each web page consists of text data, from which we extract keywords and form our index. We perform two types of queries on Wikipedia dataset: (a) *Prefix query* and (b) *Sub-string query*. For example, for a keyword *computer*, a *prefix query* can be *comp*, and a *sub-string query* can be *puter*.

For *sub-string query*, we perform document ranking based on occurrence position of the sub-string in the keywords. For example, if the keywords and their document listings (sorted in decreasing order of importance according to tf-idf scores) are:
centralization : D_1, D_2
centrifuge : D_3, D_4
century : D_5, D_6
decentralized : D_7, D_8
decent : D_9, D_{10}
where D_i's are the document identifiers.

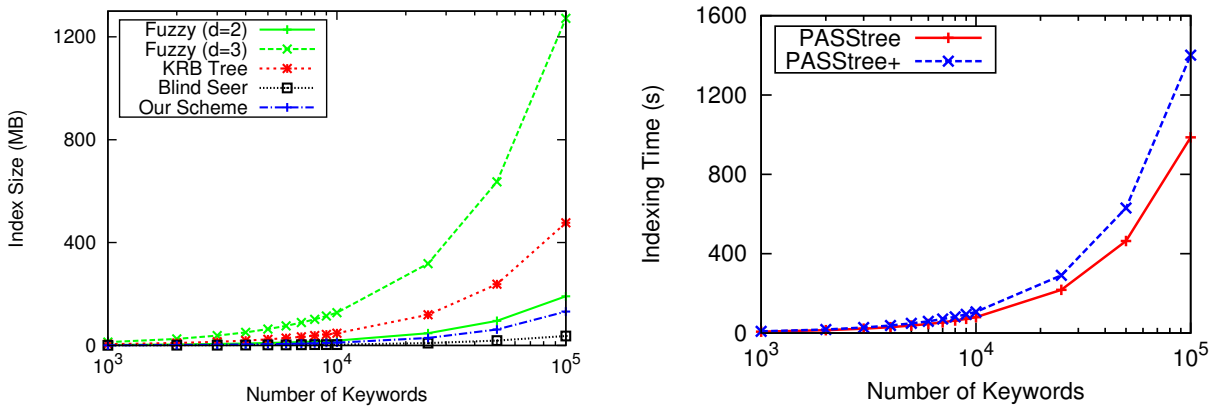
Let sub-string query be *cent*

Since *cent* occurs in first position in *centralization*, *centrifuge* and *century*, their document lists are given higher ranks. Next, *cent* occurs in third position in *decentralized* and *decent*, their document lists are given next higher ranks.
Ranked result of documents will be:
1. D_1, D_3, D_5
2. D_2, D_4, D_6
3. D_7, D_9
4. D_8, D_{10}

Thus the ranking of documents are aligned according to the occurrence position of the sub-string query in the matching keywords and also according to their relevance to the matching keywords.

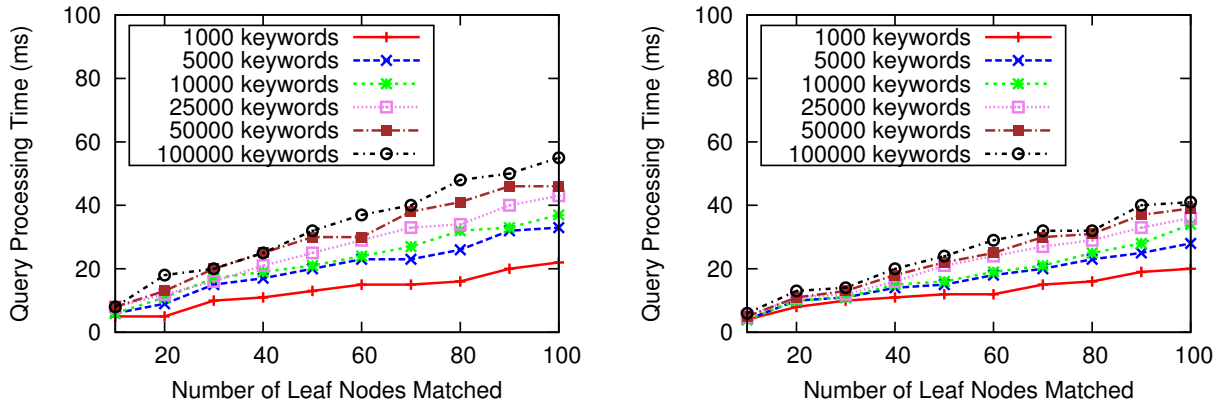
Figure 6.4(a) depicts the index size comparison of our schemes (PASStree and PASStree+) with the state-of-art practical schemes [37, 44, 27] existing in literature. The plot depicts that our scheme is memory efficient and can be used for practical scenarios like cloud. Figure 6.4(b) depicts that the index construction time is linear to the number of keywords, which is reasonable. Also, PASStree+ takes

[1]Ranking Precision is given as k'/k where k' is the top-k documents returned and k is the actual top-k documents



(a) Index Size Comparison (b) Index Construction Time

Figure 6.4 Index construction on Wikipedia Dataset



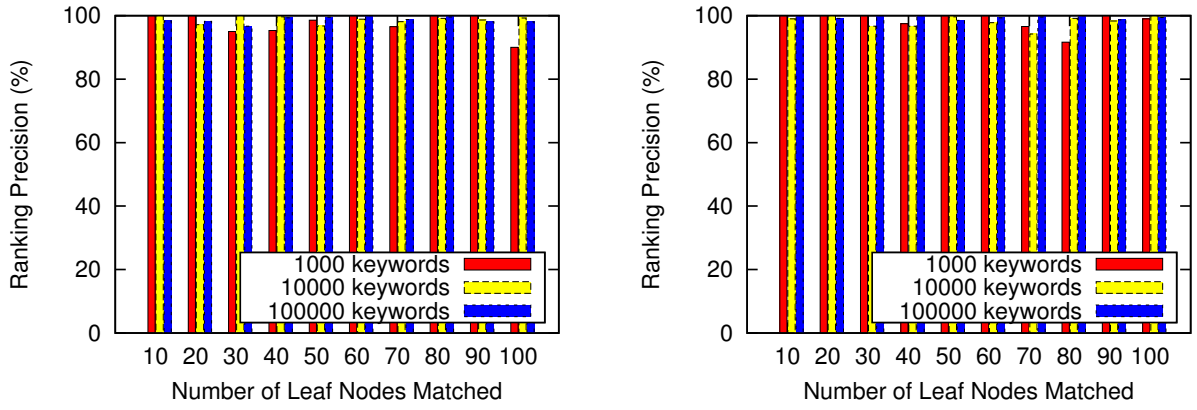
(a) PASSTree (b) PASSTree+

Figure 6.5 Query Time for prefix query on Wikipedia Dataset

more time for indexing compared to PASSTree because of extra overhead of clustering. The x-axis in in log-scale.

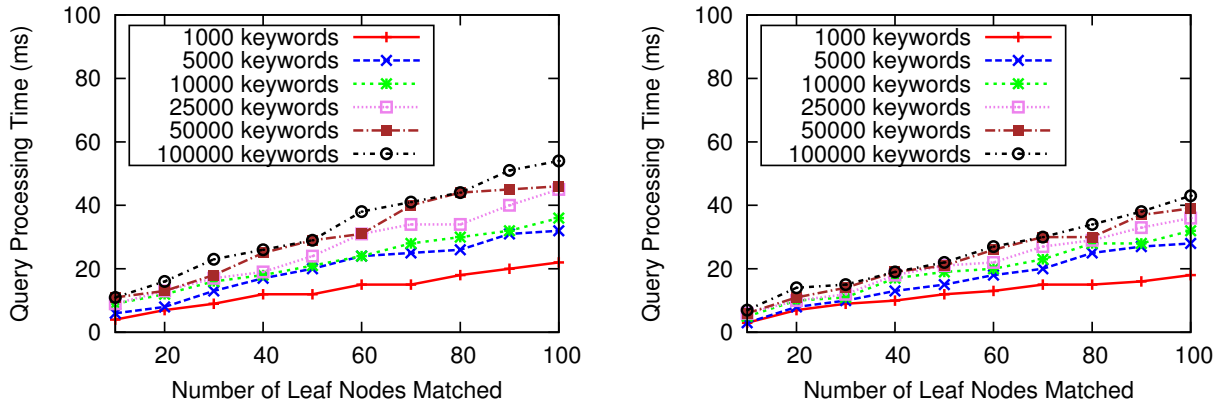
Figure 6.5(a) and Figure 6.5(b) depict the prefix query processing time of PASSTree and PASSTree+ respectively, where each individual line represents the query processing time for index built on different number of keywords. x-axis depicts the number of keywords matching the query pattern. The plots depict that PASSTree+ fares better than PASSTree in query processing, owing to the clustering of similar keywords. It is also noted that the querying takes few milliseconds for different query sizes, supporting the practicality of the schemes.

Figure 6.6(a) and Figure 6.6(b) depict the ranking precision of PASSTree and PASSTree+ respectively for prefix queries, where each individual bar represents the ranking precision for index built on different number of keywords. x-axis depicts the number of keywords matching the query pattern. The rank-



(a) PASStree (b) PASStree+

Figure 6.6 Ranking Precision for prefix query on Wikipedia Dataset



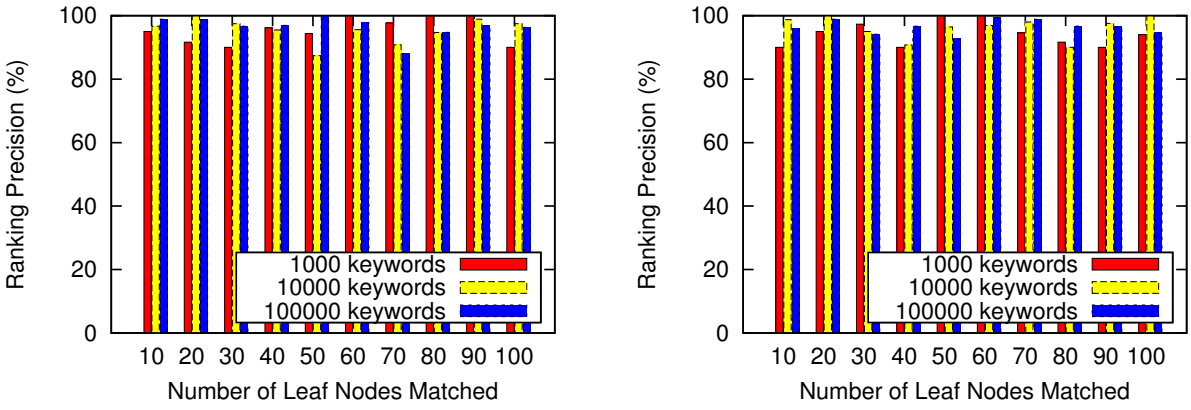
(a) PASStree (b) PASStree+

Figure 6.7 Query Time for sub-string query on Wikipedia Dataset

ing precision is calculated for top ranked documents which are returned. On an average, the ranking precision is 90% - 100%.

Figure 6.7(a) and Figure 6.7(b) depict the sub-string query processing time of PASStree and PASStree+ respectively, where each individual line represents the query processing time for index built on different number of keywords. x-axis depicts the number of keywords matching the query pattern. The plots depict that PASStree+ fares better than PASStree in query processing, owing to the clustering of similar keywords. It is also noted that the querying takes few milliseconds for different query sizes, supporting the practicality of the schemes.

Figure 6.8(a) and Figure 6.8(b) depict the ranking precision of PASStree and PASStree+ respectively for sub-string queries, where each individual bar represents the ranking precision for index built on different number of keywords. x-axis depicts the number of keywords matching the query pattern. The



(a) PASStree (b) PASStree+

Figure 6.8 Ranking Precision for sub-string query on Wikipedia Dataset

ranking precision is calculated for top ranked documents which are returned. On an average, the ranking precision is 88% - 96%.

Chapter 7

Conclusions

In this work, we presented the first approach towards string pattern matching on outsourced data in symmetric key domain. 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. Our results demonstrate the need to further explore the rich problem space of privacy preserving string pattern matching on outsourced data. 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.

Related Publications

- **Privacy Preserving String Matching for Cloud Computing**
  Bruhadeshwar Bezawada, Alex X. Liu, **Bargav Jayaraman**, Ann L. Wang and Rui Li
  In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems* (ICDCS), Columbus, Ohio, June 2015.

Bibliography

[1] B. Applebaum, Y. Ishai, and E. Kushilevitz. How to garble arithmetic circuits. *SIAM Journal on Computing*, 43(2):905–929, 2014.

[2] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 691–702. ACM, 2011.

[3] J. Baron, K. El Defrawy, K. Minkovich, R. Ostrovsky, and E. Tressler. 5pm: Secure pattern matching. In *Security and Cryptography for Networks*, pages 222–240. Springer, 2012.

[4] M. Beck and F. Kerschbaum. Approximate two-party privacy-preserving string matching with linear complexity. In *Big Data (BigData Congress), 2013 IEEE International Congress on*, pages 31–37. IEEE, 2013.

[5] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *Algorithms–ESA 2006*, pages 684–695. Springer, 2006.

[8] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):18, 2014.

[9] J. Bringer, H. Chabanne, and A. Patey. Shade: Secure hamming distance computation from oblivious transfer. In *Financial Cryptography and Data Security*, pages 164–176. Springer, 2013.

[10] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 639–648, 1996.

[11] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *Parallel and Distributed Systems, IEEE Transactions on*, 25(1):222–233, 2014.

[12] 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 *Proc. of NDSS*, volume 14, 2014.

[13] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, pages 442–455. Springer, 2005.

[14] L. Chen, X. Sun, Z. Xia, and Q. Liu. An efficient and privacy-preserving semantic multi-keyword ranked search over encrypted cloud data. *International Journal of Security and Its Applications*, 8(2):323–332, 2014.

[15] M. Chuah and W. Hu. Privacy-aware bedtree based solution for fuzzy multi-keyword search over encrypted data. In *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on*, pages 273–281. IEEE, 2011.

[16] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 79–88. ACM, 2006.

[17] E. De Cristofaro, S. Faber, and G. Tsudik. Secure genomic testing with size-and position-hiding private substring matching. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 107–118. ACM, 2013.

[18] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: An efficient and scalable protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 789–800. ACM, 2013.

[19] G. Fahrnberger. Computing on encrypted character strings in clouds. In *Distributed Computing and Internet Technology*, pages 244–254. Springer, 2013.

[20] E.-J. Goh et al. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[21] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge university press, 2009.

[22] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[23] C. Hazay and T. Toft. Computationally secure pattern matching in the presence of malicious adversaries. In *Advances in Cryptology-ASIACRYPT 2010*, pages 195–212. Springer, 2010.

[24] Y. Hua, B. Xiao, and X. Liu. Nest: Locality-aware approximate query service for cloud computing. In *INFOCOM, 2013 Proceedings IEEE*, pages 1303–1311. IEEE, 2013.

[25] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[26] A. Jarrous and B. Pinkas. Secure computation of functionalities based on hamming distance and its application to computing document similarity. *International Journal of Applied Cryptography*, 3(1):21–46, 2013.

[27] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, pages 258–274. Springer, 2013.

[28] S. Kamara and L. Wei. Garbled circuits via structured encryption. In *Financial Cryptography and Data Security*, pages 177–188. Springer, 2013.

[29] J. Katz and Y. Lindell. *Introduction to modern cryptography: principles and protocols*. CRC Press, 2007.

[30] J. Katz and L. Malka. Secure text processing with applications to private dna matching. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 485–492. ACM, 2010.

[31] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.

[32] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[33] S. Kozak, D. Novak, and P. Zezula. Secure metric-based index for similarity cloud. In *Secure Data Management*, pages 130–147. Springer, 2012.

[34] S. Kozak and P. Zezula. Efficiency and security in similarity cloud services. *Proceedings of the VLDB Endowment*, 6(12):1450–1455, 2013.

[35] K. Kurosawa and Y. Ohtaki. Uc-secure searchable symmetric encryption. In *Financial Cryptography and Data Security*, pages 285–298. Springer, 2012.

[36] M. Kuzu, M. S. Islam, and M. Kantarcioglu. Efficient similarity search over encrypted data. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1156–1167. IEEE, 2012.

[37] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou. Fuzzy keyword search over encrypted data in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5. IEEE, 2010.

[38] K. Li, W. Zhang, K. Tian, R. Liu, and N. Yu. An efficient multi-keyword ranked retrieval scheme with johnson-lindenstrauss transform over encrypted cloud data. In *Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on*, pages 320–327. IEEE, 2013.

[39] M. Li, S. Yu, N. Cao, and W. Lou. Authorized private keyword search over encrypted data in cloud computing. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 383–392. IEEE, 2011.

[40] Y. Lindell and J. Katz. Introduction to modern cryptography, 2007.

[41] J. B. Lovins. *Development of a stemming algorithm*. MIT Information Processing Group, Electronic Systems Laboratory, 1968.

[42] P. Mohassel, S. Niksefat, S. Sadeghian, and B. Sadeghiyan. An efficient protocol for oblivious dfa evaluation and applications. In *Topics in Cryptology–CT-RSA 2012*, pages 398–415. Springer, 2012.

[43] C. Örencik and E. Savaş. Efficient and secure ranked multi-keyword search on encrypted cloud data. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 186–195. ACM, 2012.

[44] 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 *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 359–374, Washington, DC, USA, 2014. IEEE Computer Society.

[45] M. Patil, S. V. Thankachan, R. Shah, W.-K. Hon, J. S. Vitter, and S. Chandrasekaran. Inverted indexes for phrases and strings. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 555–564. ACM, 2011.

[46] M. F. Porter. An algorithm for suffix stripping. *Program: electronic library and information systems*, 14(3):130–137, 1980.

[47] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[48] Z. Shao, B. Yang, and Y. Yu. Private set intersection via public key encryption with multiple keywords search. In *Intelligent Networking and Collaborative Systems (INCoS), 2013 5th International Conference on*, pages 323–328. IEEE, 2013.

[49] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[50] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.

[51] 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. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 71–82. ACM, 2013.

[52] X. Sun, X. Wang, Z. Xia, Z. Fu, and T. Li. Dynamic multi-keyword top-k ranked search over encrypted cloud data. *International Journal of Security & Its Applications*, 8(1), 2014.

[53] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik. Privacy preserving error resilient dna searching through oblivious automata. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 519–528. ACM, 2007.

[54] D. Vergnaud. Efficient and secure generalized pattern matching via fast fourier transform. In *Progress in Cryptology–AFRICACRYPT 2011*, pages 41–58. Springer, 2011.

[55] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure ranked keyword search over encrypted cloud data. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 253–262. IEEE, 2010.

[56] C. Wang, N. Cao, K. Ren, and W. Lou. Enabling secure and efficient ranked keyword search over outsourced cloud data. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1467–1479, 2012.

[57] C. Wang, K. Ren, S. Yu, and K. M. R. Urs. Achieving usable and privacy-assured similarity search over outsourced cloud data. In *INFOCOM, 2012 Proceedings IEEE*, pages 451–459. IEEE, 2012.

[58] D. Wang, S. Fu, and M. Xu. A privacy-preserving fuzzy keyword search scheme over encrypted cloud data. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 663–670. IEEE, 2013.

[59] L. Wei and M. K. Reiter. Third-party private dfa evaluation on encrypted files in the cloud. In *Computer Security–ESORICS 2012*, pages 523–540. Springer, 2012.

[60] L. Wei and M. K. Reiter. Ensuring file authenticity in private dfa evaluation on encrypted files in the cloud. In *Computer Security–ESORICS 2013*, pages 147–163. Springer, 2013.

[61] S. William and W. Stallings. *Cryptography and Network Security, 4/E*. Pearson Education India, 2006.

[62] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[63] A. C.-C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[64] M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*, pages 65–76. ACM, 2013.

[65] B. Zhang, R. Cheng, and F. Zhang. Secure hamming distance based record linkage with malicious adversaries. *Computers & Electrical Engineering*, 2013.

[66] Y. Zhang and C. Zhang. Secure top-k query processing via untrusted location-based service providers. In *INFOCOM, 2012 Proceedings IEEE*, pages 1170–1178. IEEE, 2012.

[67] Y. Zhu, R. Xu, and T. Takagi. Secure k-nn computation on encrypted cloud data without sharing key with query users. In *Proceedings of the 2013 international workshop on Security in cloud computing*, pages 55–60. ACM, 2013.

[68] X. Zou, Y.-S. Dai, and E. Bertino. A practical and flexible key management mechanism for trusted collaborative computing. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008.