A New Lightweight Symmetric Searchable Encryption Scheme for String Identification

Indranil Ghosh Ray, Yogachandran Rahulamathavan and Muttukrishnan Rajarajan, Senior Member, IEEE

Abstract—In this paper, we provide an efficient and easy-to-implement symmetric searchable encryption scheme (SSE) for string search, which takes one round of communication, $O(n)$ times of computations over $n$ documents. Unlike previous schemes, we use hash-chaining instead of chain of encryption operations for index generation, which makes it suitable for lightweight applications. Unlike the previous SSE schemes for string search, with our scheme, server learns nothing about the frequency and the relative positions of the words being searched except what it can learn from the history. We are the first to propose probabilistic trapdoors in SSE for string search. We provide concrete proof of non-adaptive security of our scheme against honest-but-curious server based on the definitions of [12]. We also introduce a new notion of search pattern privacy, which gives a measure of security against the leakage from trapdoor. We have shown that our scheme is secure under search pattern indistinguishability definition. We show why SSE scheme for string search cannot attain adaptive indistinguishability criteria as mentioned in [12]. We also propose modifications of our scheme so that the scheme can be used against active adversaries at the cost of more rounds of communications and memory space. We validate our scheme against two different commercial datasets (see [1], [2]).

Index Terms—Cloud storage, Symmetric key, Searchable encryption, hash-chain, lightweight cryptography.

I. INTRODUCTION

The cloud is designed to hold a large number of encrypted documents. With the advent of cloud computing, growing number of clients and leading organizations have started adapting to the private storage outsourcing. This allows resource constrained clients to privately store large amounts of encrypted data in cloud at low cost. However, this prevents one from searching. This gives rise to a newly emerging field of research, called searchable encryption (SE). SE can be classified into symmetric searchable encryptions (SSE) and asymmetric searchable encryptions (ASE). In this paper, we study the SSE for string search. In the SSE, the client encrypts the data and stores it on the cloud. It may be noted that client can organize the data in an arbitrary manner and can maintain additional data structures to achieve desired data efficiency. In this process, the initial client side computation is thus as large as the data, but subsequent computations to access data is less for both client and the cloud server.

Since huge volumes of documents are stored in a cloud server, searching against a keyword may result into large number of documents, most of which are not intended, causing unnecessary network traffic. This motivates the idea of searching against a string, which allows the search to be more specific. Searching for string is a multi-keyword search where the ordering of keywords is preserved. So in addition to the presence of all these keywords in a document, their ordering and adjacency are to be taken care off while searching. So the index table needs to be prepared in such a way that the adjacency information of the words can be preserved.

Although few works are available in the literature involving string search (e.g. [15], [18], [22], [23], [25]), but most of them lack formal security proof against the revised definitions of [12] and also expose lots of informations to the server following the search (see Table I of Section [1]. In the SSE scheme, the server is expected to learn nothing about the search queries and data collections. SSE achieves this by using symmetric cryptographic primitives instead of heavy computations of public key encryption at the cost of small leakage of information [12]. Here we take an example which will be extended throughout the paper to illustrate our algorithms and data structures.

Example 1: Let us consider the text file ToyExample.txt in the following figure, which is to be encrypted and uploaded to a public cloud for future search.

ToyExample.txt

This is a demonstration text for showing how line breaking works.

Let us consider the set of keywords as follows : keywords = {this, is, a, demonstration, text, for, showing, how, line, works, breaking}.

The client encrypts and outsources this file to the server and later he wants to search a string “this is a demonstration”.

Remark 1. While generating the index, client converts all letters into lower case and generate the index and trapdoors accordingly.

It may be noted that in traditional SE schemes, this is treated as multi-keyword search for keywords : this, is, a, and demonstration. The drawback of this approach is that the adjacency of the words is not considered. However, the proposed string search not only looks for those keywords, but also consider the order. We will continue using this example in the subsections of Section IV to explain different phases of our proposed scheme.

Our Contributions : In [12], authors proposed the first efficient SSE construction, achieving sublinear search time and introduced the notion of non-adaptive and adaptive indistinguishability definitions of security for SSE. In the same work, authors introduced the idea of history connected to a finite number of consecutive keyword searches. We extended that definition for string search as is necessary for security proof, and call it history-of-strings. Using this new definition, we carryout the necessary changes in the definition of non-adaptive indistinguishability [12] for SSE performing string search. Finally we prove that our proposed scheme is secure under the non-adaptive indistinguishability definition of SSE security against honest-but-curious server. Although indistinguishability definition of SSE security takes care of the the security of keyword from index, however it does not provide security against the leakage from trapdoor. Towards this we have introduced the notion of search pattern security and have shown our scheme to be secure under search pattern indistinguishability definition. The novelty of our scheme is
that although the index is generated by the client at the beginning, and remains same for the same dataset through out the process and thus static in nature, however the trapdoors are dynamic in nature, making it more difficult for the eavesdroppers to understand the search patterns and thus is more secure against attacks like replay attacks, frequency analysis based attacks and many more.

Our scheme achieves searches in one communication round and requires \(O(n)\) times computations for searching a string in \(n\) documents, which is optimal. Also the scheme requires no storage on the client side and \(O(n)\) storage on the server side for the \(n\)-document collection. Lastly, the scheme guarantees minimal leakage in a sense that server directly knows nothing about the frequency of the words being searched and their relative positions in the documents except what it can learn from the history of search. Unlike the index generation techniques (sequence of encryptions of a key) used in [12], [15], we use the hash-chain technique, which is faster, and is thus suitable for lightweight applications.

For the first time we address the problem of string search using symmetric searchable encryption against the active adversary, who by trick can place a document of his choice in the document collections. We propose a modification of our scheme to deal with active adversary securely at the cost of maintaining a list of keywords at the client’s end and two rounds of communications.

We also implement the scheme against two different commercial datasets, namely, a 20 MB DNA dataset [2] and a 19 MB TIMIT speech data [1] and successfully achieve string search functionality in encrypted domain.

Rest of the paper is organized as follows: In Section II we discuss the related works in SSE. In Section III we provide definitions and preliminaries. In Section IV we discuss our proposed scheme in detail. In Section V we analyze the security of our proposed scheme. In Section VI we discuss the time and space complexity of our scheme and provide experimental results of our scheme against commercial datasets (see [1], [2]). We conclude the paper in Section VII.

II. RELATED WORKS

For the last ten years, searchable encryption has been the focus for many leading research groups and several results were proposed [3]–[10], [12], [16], [7], [19], [20], [24]. In [3], authors defined computational and statistical relaxations of the existing notion of perfect consistency and provided a new scheme that was statistically consistent. They also proposed a transformation of an anonymous identity based encryption scheme (IBE) to a secure public key encryption with keyword search scheme (PEKS) that guarantees consistency. In [4] authors presented as-strong-as-possible definitions of privacy and some constructions for public-key base encryption schemes where the encryption algorithm is deterministic. In the same work, new methods were proposed for database encryption that permit fast (i.e. sub-linear, and in fact logarithmic, time) search while provably providing privacy that is as strong as possible subject to this fast search constraint. The work in [4] also generalizes their methods to obtain a notion of efficiently searchable encryption schemes which permit more flexible privacy to search-time trade-offs via a technique called bucketization. In [5], authors studied the problem of searching on data that is encrypted using a public key system which they referred as PEKS and provided several constructions. In [6], authors show how to create a public-key encryption scheme that allows PIR (private information retrieval) searching over encrypted documents. Their solution was the first to reveal no partial information regarding the users search (including the access pattern) in the public-key setting and with small communication complexity. In [7], authors defined and solved the problem of privacy-preserving multi-keyword ranked search over encrypted data in cloud computing (MRSE). They established a set of strict privacy requirements for such a secure cloud data utilization system. Among various multi-keyword semantics, they choose the efficient similarity measure of “coordinate matching,” i.e., as many matches as possible, to capture the relevance of data documents to the search query. They also used “inner product similarity” to quantitatively evaluate such similarity measure. They provide two MRSE schemes to achieve various stringent privacy requirements in two different threat models. In [24], authors proposed an efficient searchable encryption scheme for auction (SESA) in emerging smart grid marketing, which is based on a public key encryption with keyword search technique to enable the energy sellers to inquire suitable bids while preserving the privacy of the energy buyers. In [6] authors provided a systematic study of various attack models against SSE based schemes.

Dynamic SSE was first considered by Song et al. [19], but no solution with sublinear search time existed before the work of Kamara et al. [13]. Recently, two new dynamic SSE schemes have been proposed. The first one, by Cash et al. [9], which is an extension of [10]. They showed that SSE is feasible on very large databases. In [9], authors designed and implemented dynamic symmetric searchable encryption schemes that efficiently and privately search server-held encrypted databases with tens of billions of record-keyword pairs. Their basic theoretical construction was for single-keyword searches and which offers asymptotically optimal server index size, fully parallel searching, and minimal leakage. In [10], authors presented another efficient SSE scheme which supports complex queries involving multiple keywords. Similar scheme may be found in [17]. In [11], authors studied the trade-off between locality and server storage size of SSE schemes.

In [12], authors introduced the idea of SSE with improved security definitions. They introduced the two most important security definitions, namely non-adaptive indistinguishability and adaptive indistinguishability. They also proposed SSE schemes for keyword search which they proved to be secure under these new security definitions.

In [15], authors studied the security provided by various encrypted databases and presented a series of attacks that recover the plaintext from encrypted database columns using only the encrypted column and publicly-available auxiliary information. In [17], authors studied efficient sub linear search techniques for arbitrary Boolean queries. They considered scalable DBMS with provable security for all parties, including protection of the data from both server (who stores encrypted data) and client (who searches it), as well as protection of the query, and access control for the query.

In [20], Stefanov et al. designed scheme for the first time to address forward secrecy. However, the problem of malicious servers has not been studied, except in [20], but, as we will see later, their proposition is flawed. In [19], the search complexity is linear in the number of documents stored in the database.

In [13], [18], [22], [23], [25], SSE schemes are developed for string search. In [22], authors designed the first SSE scheme for phrase search. This scheme works in two phases, each taking one round of communication. In the first phase these documents are identified which contains all words occurring in the phrase. In the second round the candidate documents are checked to confirm the existence of the phrase. In [15], authors proposed a scheme for string search in non-adaptive setting where they used some additional data structures and techniques (list, lookup tables, pseudo random functions and hash-chains for word sequencing) to keep track of position informations. In [15] the index generation technique is similar to the index generation of [12] and requires a sequence of encryption operation while forming index. In this paper, we achieve
the same non-adaptive security by using a sequence of hashing instead of encryption operations which is faster and suitable for lightweight applications. In [23] schemes are proposed that enables efficient searching for an arbitrary string that may not be extracted as keywords at the cost of leaking some information for the sake of efficiency. In [25], authors introduced a SSE scheme that allows both encrypted phrase searches and proximity ranked multi-keyword searches to encrypted datasets on untrusted cloud. In [15], authors propose a faster way of secure string search based on bloom filters. It may be noted that our approach is based on index based scheme and we prove it to be non adaptively secure according to the definition introduced in [13].

Here we summarize our contributions:

1. We propose a non-adaptively secure SSE scheme for string search which takes one round of communication, \(O(n)\) times of computation over \(n\) documents, \(O(n)\) additional memory in the server side and no memory in the client side.
2. We provide a formal and proof to show that the scheme is non-adaptively secure.
3. Although schemes of [12] is secure against non-adaptive indistinguishability definition of security, but they does not guarantee security against the leakage from trapdoor. Unlike the previous SSE schemes, this is the first SSE scheme for string search which generates probabilistic trapdoors which allows search pattern privacy. We formally introduce this new notion of security in SSE and show that our scheme is secure under search pattern indistinguishability definition (see Definition 2).
4. In the previous SSE schemes for string search, the index tables are generated by creating linked-lists corresponding to every keywords, where informations related to occurrence of the keyword in i-th document is stored in i-th node along with the key, say \(k_{i+1}\), and is encrypted with a key, say, \(k_{i-1}\). Which leads to a sequence of encryption functions for generating the index and a sequence of decryption functions while searching. We achieve the same non-adaptive security by applying sequence of masked hashing, i.e., hash-chain, which is faster and suitable for lightweight applications.
5. We propose modifications of our scheme so that it can be used against active adversary.

III. NOTATIONS AND DEFINITIONS

**Document collections and Data Structures:** Let \(\Delta = \{w_1, w_2, \ldots, w_d\}\) be a dictionary of \(d\) words and \(P(\Delta)\) be the set of all possible documents which are collections of words. Let \(D \subseteq \mathbb{P}(\Delta)\) be the collection of \(n\) documents \(D = \{D_1, D_2, \ldots, D_n\}\). Let \(id(D_i)\) be the unique identifier for the document \(D_i\). We denote the list of all \(n\) document identifiers in \(D\) by \(id(D)\), i.e., \(id(D) = \{id(D_1), \ldots, id(D_n)\}\). Furthermore, let \(D(w_j)\) be the collection of all documents in \(D\) containing the word \(w_j\). A string \(s\) of \(l\) words is an ordered tuple \((w_1, w_2, \ldots, w_l)\). Let \(D(s)\) denotes a collection of documents in \(D\) that contains the string \(s\). It is easy to check that \(D(s) \subseteq \bigcup_{j=1}^{l} D(w_j)\).

We denote by \(\delta(D)\), all the distinct keywords connected to the document collection \(D\). We denote by \(f_j\), the frequency of occurrence of the keyword \(w_j\) in \(D\). Let \(f_j = \max\{f_j[i] | 1 \leq i \leq n\}\). Also let \(f = \max\{f_j[j] | 1 \leq j \leq \delta(D)\}\).

Since the index is meant for server to perform search, we call it server side index and denote it by SI. The generation of SI involves generation of index table \(I\) along with the vectors \(I_t, I_c, I\). \(I_c\) is a \(n \times \delta(D)\) array where i-th row corresponds to i-th document \(D_i\) and j-th column corresponds to the word \(w_j\) and \((i,j)\)-th element is denoted by \(I[i][j]\). \(I[i][j]\) is a set of \(f\) number of \(\lambda\)-bit strings treated as element of \(Z_\lambda\), where \(p\) is here a \(\lambda\)-bit prime, \(\lambda\) being the security parameter.

**Cryptographic Primitives:** Here we define cryptographic primitives that are needed for our SSE scheme for string search.

**Definition 1.** A symmetric key encryption scheme is a probabilistic polynomial-time algorithms \((\text{Gen}, \text{Enc}, \text{Dec})\) such that:

1. The key-generation algorithm \(\text{Gen}\) takes as input the security parameter \(\lambda^k\) and outputs a key \(k\); we write this as \(k \leftarrow \text{Gen}(1^\lambda)\) (thus emphasizing the fact that \(\text{Gen}\) is a randomized algorithm). We will assume without loss of generality that any key \(k\) output by \(\text{Gen}(1^\lambda)\) satisfies \(|k| > \lambda\).
2. The encryption algorithm \(\text{Enc}\) takes as input a key \(k\) and a plaintext message \(m \in \{0, 1\}^*\), and outputs a ciphertext \(c\). Since \(\text{Enc}\) is randomized, we write this as \(c \leftarrow \text{Enc}_k(m)\).
3. The decryption algorithm \(\text{Dec}\) takes as input a key \(k\) and a ciphertext \(c\), and outputs a message \(m\). We assume that \(\text{Dec}\) is deterministic, and so write this as \(m = \text{Dec}_k(c)\). It is required that for every \(\lambda\), every key \(k\) output by \(\text{Gen}(1^\lambda)\), and every \(m \in \{0, 1\}^*\), it holds that \(\text{Dec}_k(\text{Enc}_k(m)) = m\).

We typically denote an arbitrary negligible function by \(\text{negl}\) such that for any arbitrary polynomial \(p(\lambda)\), there exists an integer \(a\) such that for all \(\lambda > a\), \(\text{negl}(\lambda) < \frac{1}{p(\lambda)}\).

**Definition 2.** private-key encryption scheme \(\pi = (\text{Gen}, \text{Enc}, \text{Dec})\) with indistinguishable encryptions under a chosen-plaintext attack (or is IND-CPA secure) if for all probabilistic polynomial-time adversaries \(A\) there exists a negligible function \(\text{negl}\) such that \(\Pr[\text{PrivK}^{\text{mcpa}}_{\pi}(\lambda) = 1] \leq \frac{1}{2} + \text{negl}(\lambda)\), where the probability is taken over the random coins used by \(A\), as well as the random coins used in the experiment (for choosing the key, the random bit \(b\), and any random coins used in the encryption process).

Throughout the paper we use the encryption and decryption functions \((\text{Enc}_k()\) and \(\text{Dec}_k()\) which are from IND-CPA symmetric encryption scheme \(\pi\). We also use pseudo prime number generator \(\text{Gen}(1^\lambda)\), denoted by \(PPNG(1^\lambda)\) which outputs a \(\lambda\)-bit probabilistic prime number. In addition, we use message authentication code, \(\text{MAC}_k()\) which outputs in \(\lambda\) bits. We treat these outputs as elements of \(Z_\lambda\), where \(p\) is a \(\lambda\) bit prime. We write them simply as \(\text{MAC}_k()\) and \(\text{Enc}_k()\). Also we use \(\oplus, \odot\) and \(\otimes\) as the addition, subtraction and multiplication operators in \(Z_\lambda\). Also for \(a \in Z_\lambda\), \(a^{-1}\) denotes the multiplicative inverse in \(Z_\lambda\).

Before closing this section we provide in Table II the list of notations discussed in this section.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>String search</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>non-adaptive security</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>adaptive security</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>security against active adversary</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>client storage</td>
<td>no</td>
<td>dictionary</td>
<td>trusted server</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>no of rounds</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>storage cost</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>no. of encryptions per keyword</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>


### Table II. Summary of notations.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>document collection.</td>
</tr>
<tr>
<td>$D_i$</td>
<td>$i$-th document.</td>
</tr>
<tr>
<td>$n$</td>
<td>number of documents.</td>
</tr>
<tr>
<td>$id(D_i)$</td>
<td>the unique identifier for the document $D_i$.</td>
</tr>
<tr>
<td>$D(w_i)$</td>
<td>list of document identifiers in the document collection.</td>
</tr>
<tr>
<td>$D(s)$</td>
<td>collection of all documents in $D$ containing all words in the string $s$.</td>
</tr>
<tr>
<td>$N(D)$</td>
<td>all the distinct keywords connected to the document collection $D$.</td>
</tr>
<tr>
<td>$f$</td>
<td>the frequency of occurrence of the keyword $w_i$ in $D_i$.</td>
</tr>
<tr>
<td>$I_{f_i,j}$</td>
<td>$max{f_j</td>
</tr>
<tr>
<td>$I_{f_i}$</td>
<td>contains list of inverted indexes for the list of keywords.</td>
</tr>
<tr>
<td>$I_r$</td>
<td>$r$ contains list of encrypted document identifiers.</td>
</tr>
<tr>
<td>$SI$</td>
<td>server side index which is a triplet $(I_r, I_c, I_e)$.</td>
</tr>
</tbody>
</table>

### IV. Our Scheme

In this section we present our SSE scheme $\Pi_{ss}$ for string search, which is composed of four algorithms KeyGen, BuildIndex, Trapdoor and Search. First we formally define the scheme. In the subsequent subsections, we will discuss these algorithms in detail with illustrations.

#### Scheme 1 ($\Pi_{ss}$).

The scheme $\Pi_{ss}$ is a collection of four polynomial time algorithms (KeyGen, BuildIndex, Trapdoor, Search) such that:

1. **KeyGen($1^k$)**: KeyGen is a probabilistic key generation algorithm that is run by the client to setup the scheme (see Algorithm 1). It takes a security parameter $\lambda$, and returns a secret master key $k_m$ and a mask-key $k_r$ which are to be kept privately at client’s end and a session key $k_s$ which is to be shared between client and the server. Client also shares a $\lambda$-bit prime $p$ with the server. The length of $k_m$, $k_r$ and $k_s$ are polynomially bounded in $\lambda$.

2. **BuildIndex($k_m, k_r, k_s, p$)**: BuildIndex is a probabilistic algorithm run by the client to generate $SI = (I_r, I_c, I_e)$. It takes $k_m$, $k_r$, $k_s$, $p$ and returns SI. Since BuildIndex is randomized, we write this as $SI \leftarrow \text{BuildIndex}^k(k_m, k_r, k_s, p)$ (see Algorithm 2).

3. **Trapdoor($k_m, k_r, k_s, p, s$)**: Trapdoor is a probabilistic algorithm run by the client to create a trapdoor for a given string of words $s = (w_1, w_2, \ldots, w_l)$. It takes $k_m, k_r, k_s, p$ and $s$ as input and outputs $t = (t_1, t_2, \ldots, t_l)$, where $t_i$ is the trapdoor corresponding to the word $w_i$. Since trapdoor is randomized, we write this as $t \leftarrow \text{Trapdoor}^k(k_m, k_r, k_s, p)$ (see Algorithm 3).

4. **Search($SI, t$)**: Search is run by the server in order to search for the documents in $D$ that contain the string $s$. It takes $k_m$, $SI$ and trapdoor $t$ of the string $s$ as inputs, and returns $D(s)$, the set of identifiers of documents containing the string $s$. Since this algorithm is deterministic, we write it as $D(s) = \text{Search}^k_s(SI, t)$ (see Algorithm 4).

#### Structure of index:

In the array $I$, informations are stored related to positions of a word in a document. Recall that $I[i][j]$ is a set of $f$ number of $\lambda$-bit strings. Now we describe how these bit strings are obtained. To keep track of ordering of words in a document, instead of position pointers, we use hash-chain. For a $l$-word sentence $(w_1, \ldots, w_l)$ in a document, say $D_i$, we first take a random integer $r \in Z_p$. Then we form a hash-chain $(r_1, \ldots, r_l)$ such that $r_1 = r$ and for $2 \leq j \leq l$, $r_j = MAC_{s_k}(r_{j-1})$. Then we put $r_{j-1} + msk_j$ in $I[i][j]$, where $msk_j = MAC_{s_k}(w_j)$. It may be noted that since $k_m$ is privately kept at client’s end, server cannot compute the mask. The masks are provided to the server through the trapdoors. So for each time the word $w_j$ occurs in $D_i$ some integer from $Z_p$ is generated as mentioned above and sorted in $I[i][j]$. Since $I[i][j]$ can then reveal the frequency of the word $w_j$ in $D_i$, we insert additional $(f - |I[i][j]|)$ number of random numbers from $Z_p$ in each $I[i][j]$ so that $|I[i][j]|$ becomes $f$ for all $i$’s and $j$’s. We call it cell-padding. This stops the leakage informations related to frequency and relative positions completely.

We call $MAC_{k_m}(w_j)$ and index or client index of the word $w_j$, $I[j]$ contains inverted index of the word $w_j$, which is a $\lambda$-bit string. $I[i]$ is encrypted identifier of $D_i$, where the key of encryption is $k_m$.

#### Secure update of index:

Our index table is dynamic in a sense that for each update of document collection the index can be updated without disturbing the existing index just by adding a new row corresponding to the new document where the entries in index are computed as described above.

In the following subsections, we provide the algorithms for KeyGen, BuildIndex, Trapdoor and Search in detail with illustrations.

#### A. Key Generation

**Algorithm 1 Keygen**

**Input** security parameter $\lambda$.

**Output** $k_m, k_r, k_s$ and $p$.

$k_m, k_r, k_s \leftarrow \text{Gen}^k(1^\lambda)$;

$p \leftarrow \text{PPNG}(1^\lambda)$;

#### B. Index Generation

**Algorithm 2 BuildIndex**

**Input** $k_m, k_r, k_s, p, D = (D_1, \ldots, D_n)$.

**Output** $SI = (I_r, I_c, I_e)$.

Form a collection $W = \{w_1, \ldots, w_d\}$ of all distinct words occurring in $D$;

$j \leftarrow 1$;

while $j \leq d$ do

\[c_{i_j} = MAC_{k_m}(w_j)\];

\[I[i][j] = c_{i_j}^{-1};\]

\[j \leftarrow j + 1;\]

end while;

while $i \leq n$ do

\[I[i][j] = Enc_{k_m}(id(D_i));\]

For each sentence $s = (w_{i1}, \ldots, w_{ik})$ in $D_i$, chose $r \in Z_p$ randomly and form $(r_1, \ldots, r_l)$ such that $r_1 = r$ and for $2 \leq j \leq l$, $r_j = MAC_{s_k}(r_{j-1})$. Associate $r_j$ with the word $w_{ij}$;

\[j \leftarrow 1;\]

while $j \leq d$ do

set $I[i][j]$ as all integers in $Z_p$ that are associated with the word $w_j$ and add the mask $m_j = MAC_{s_k}(w_j)$ with all of them in modulo $p$, i.e., in $Z_n$;

if $(|I[i][j]| < f)$ then

Inject $(f - |I[i][j]|)$ number of random elements from $Z_p^*$ in $I[i][j]$;

end if;

\[j \leftarrow j + 1;\]

end while;

\[i \leftarrow i + 1;\]

end while;
Example 1 (continued): (Illustration of BuildIndex)
Let the prime number be 31, i.e., \( p = 31 \). Also let us assume that \( MAC_{k_m}(this) = 18 \), \( MAC_{k_m}(\text{demonstration}) = 9 \), \( MAC_{k_m}(a) = 13 \), \( MAC_{k_m} \) (text) = 15, \( MAC_{k_m} \) (for) = 10 and \( MAC_{k_m} \) (is) = 6. It may be noted that the first row of server index table (see Table III), i.e., \( I_r \) represents inverses modulo 31 of client index values. For example \( MAC_{k_m}(this) \mod 31 = 18 \mod 31 = 19 \). Thus the column corresponding to 19 is for the keyword “this”. Similarly other values in the first rows are computed corresponding to other keywords. Let the search query be “This is a demonstration”. Now we demonstrate how BuildIndex algorithm generates a portion of server side index corresponding to this search query occurring in the document ToyExample.txt. Let the encrypted file identifier corresponding to ToyExample.txt be 1221. For the word ‘this’, first we select \( r_1 = 10 \) \( \equiv Z_p \). So the hash chain corresponding to the search query “This is a demonstration” is \((10, MAC_{k_m}(10), MAC_{k_m}(10), MAC_{k_m}(10))\) and let this gives the sequence \((10, 21, 5, 16)\). So when masking is applied to this sequence, we get \((10 + MAC_{k_m}(this), 21 + MAC_{k_m}(is), 5 + MAC_{k_m}(a), 16 + MAC_{k_m}(demonstration))\) and let this gives the sequence \((21, 21, 30, 4)\). Table III presents the server side index without cell-padding (see the paragraph “Structure of index” in Section IV).

### Table III

<table>
<thead>
<tr>
<th>I</th>
<th>19</th>
<th>26</th>
<th>12</th>
<th>7</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>1221</td>
<td>2</td>
<td>21</td>
<td>30</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Remark 2. In [13], authors have proposed a SSE scheme for string search, which is similar to the non-adaptive SSE scheme of [12] for keyword search with some additional data structures and techniques (list, lookup tables, pseudo random functions and hash-chains for word sequencing) being used to keep track of position informations. However, with this approach, server learns the word frequency and relative positions of the underlying document. The BuildIndex algorithm (see Algorithm 3) is based on a new approach of inverted index generation in modulo prime field. As opposed to the unmasked hash-chains used in [13] for all words in a document, we use masked hash-chain and cell-padding which stops leakage of informations related to the relative positions of sentences and the frequency of words. As opposed to the idea of chain of encryptions in [12], we introduce the idea of masking for the security of index which is faster. In earlier schemes, to search for a word in \( n \) documents, \( n \) decryption operations were needed. In our scheme, all we need is unmasking which is a subtraction operation in \( Z_p \) for all entries of the corresponding column.

C. Trapdoor Generation

Example 1 (continued): (Illustration of Trapdoor)
Let the search query be “This is a demonstration”. Let us consider the computation of \( Trapdoor(this) = (t_1, t_2, t_3) \) as follows:
\( Enck_m(this) = e = 20 \) and \( MAC_k(e) = msk = 24 \); We know \( MAC_k(this) = ci = 18 \). Now, let \( t_1 = MAC_k(e \oplus msk \oplus ci) = MAC_k(62) = 14 \). Also let \( t_2 = e \oplus ci = 20 \times 18 = 19 \). Lastly, let \( t_2 = e \oplus msk = 20 \times 24 = 15 \). Thus \( Trapdoor(this) = (t_1, t_2, t_3) = (14, 19, 15) \). Similarly we compute trapdoors for the key words ‘is’, ‘a’ and ‘demonstration’.

### Algorithm 3 Trapdoor

**Input** \( w = (w_{c_1}, w_{c_2}, \ldots, w_{c_5}) \), \( k_m \), \( k \), \( p \), \( MAC_k(\cdot) \).
**Output** \( t = (t_1, \ldots, t_l) \).

\[
\begin{align*}
& j \leftarrow 1; \\
& \text{while } j \leq l \text{ do} \\
& \quad e = Enck_m(w_{c_j}); \\
& \quad msk = MAC_k(e); \\
& \quad ci = MAC_k(w_{c_j}); \\
& \quad t_{j_1} = MAC_k(e \oplus msk \oplus ci); \\
& \quad t_{j_2} = e \oplus ci; \\
& \quad t_{j_3} = e \oplus msk; \\
& \quad t_j = (t_{j_1}, t_{j_2}, t_{j_3}); \\
& \quad j \leftarrow j + 1; \\
& \text{end while}
\end{align*}
\]

**Remark 3.** Although the index table is static, i.e., created once at the beginning, but for every instance of search we compute probabilistic trapdoors and using these, we end up searching successfully from the index table. It may be noted that the probabilistic algorithm used in the BuildIndex is only for obtaining encrypted file pointers. Once the keys are fixed, \( I_r[j]'s \) and the entries in \( I_l[i]'s \) before cell-padding are deterministic. Let \( t = (t_{j_1}, t_{j_2}, t_{j_3}) \) be the trapdoor corresponding to the word \( w_j \) and \( I_r[j] \) contains the corresponding inverted client index. So \( I_l[j] = ci^{-1} \mod 2 \), where \( ci = MAC_k(w_{c_j}) \). Note that \( t_{j_2} = e \times ci \), where \( e \) is the random part which is obtained only when \( t_{j_2} \) is multiplied with \( I_l[j] \). Once \( e \) is obtained, \( msk \) can be obtained as \( msk = t_{j_2} \times e^{-1} \). The correctness is asserted from the check \( MAC_k(e \oplus msk \oplus ci) = t_{j_1} \).

D. Searching

### Algorithm 4 Search

**Input** \( t = (t_1, \ldots, t_l) \), \( SI \), \( k_m \), \( MAC_k(\cdot) \).
**Output** \( encrypted_file_pointers \), a list of encrypted document pointers:
the list column and column_mask are set empty;
\( i \leftarrow 1; \)

\[
\begin{align*}
& \text{while } i \leq l \text{ do} \\
& \quad j \leftarrow 1; \\
& \quad \text{while } j \leq d \text{ do} \\
& \quad \quad e = (t_{j_2} \oplus I_l[j]); \\
& \quad \quad m = t_{j_3} \oplus e^{-1}; \\
& \quad \quad \text{if } (MAC_k(e \oplus m \oplus I_l[j]^{-1}) = t_{j_1}) \text{ then} \\
& \quad \quad \quad \text{set mask of } j \text{ th column as } msk = m, \text{ add } j \text{ to column} \\
& \quad \quad \quad \text{ and } msk \text{ to column_mask;}
\end{align*}
\]

\[
\begin{align*}
& \text{end if} \\
& \quad j \leftarrow j + 1; \\
& \text{end while} \\
& i \leftarrow i + 1; \\
\end{align*}
\]

\[
\begin{align*}
& \text{end while} \\
& \text{the list } encrypted_file_pointers \text{ is set empty;} \\
& i \leftarrow 1; \\
& \text{while } i \leq n \text{ do} \\
& \quad \text{if } (\text{there exists } l \text{ integers } p_1, \ldots, p_l \text{ such that } (p_j \oplus \text{column_mask[j]}) \in \{i\}[\text{column[j]}], 1 \leq j \leq l \text{ and } MAC_k(p_j) = p_{j+1} \text{ for } 1 \leq j \leq l - 1) \text{ then} \\
& \quad \quad \text{add } I_l[i] \text{ to } encrypted_file_pointers. \\
& \end if \\
& \quad i \leftarrow i + 1; \\
& \text{end while}
\end{align*}
\]
Example 1 (continued): (Illustration of Searching)

After retrieving the trapdoor for the word ‘this’, server identifies the column in the server index table corresponding to the word ‘this’ in the following way:

It may be noted that when the column index is 1, we have \((t_2 \otimes \ell_{[1]}) = 19 \otimes 19 = 20 = e\). Also \(t_3 \otimes e^{-1} = 24 = msk\). Thus \(MAc_k (e \otimes msk \otimes L_{[j]}^{-1}) = MAc_k (62) = 14 = t_1\). Thus the column corresponding to index 1 is identified for the trapdoor (14, 19, 15) which was sent for the keyword ‘this’. Similarly the column indexes are identified for the other keywords which are 2, 3 and 4 for the words ‘is’, ‘a’ and ‘demonstration’ respectively. Note that server can retrieve mask corresponding to keyword ‘this’ which is \(msk = 24 = t_3 \otimes e^{-1}\). Similarly server can compute masks for other words as well. So from the sequence obtained from the server index table, i.e., \((2, 21, 30, 4)\), server can compute the sequence \((10, 21, 5, 16)\) by subtracting corresponding masks. Now server forms the hash chain \((10, 21, 5, 16) = (10, MAc_k (10), MAc_k^2 (10), MAc_k^3 (10))\) which tells that this string exists in the document whose encrypted identifier is 1221. Server returns this to client which on decryption yield ToyExample.txt, i.e. the filename containing the queried string.

Remark 4. In Algorithm 2 the input \((t_1, \ldots, t_l)\) corresponds to the l-word search string. In the index table, there are \(|\delta(D)|\) number of columns, out of which the l columns corresponding to \(t_1, \ldots, t_l\) are to be identified and also the corresponding column masks are to be computed for searching. This is done in the first while loop of the search algorithm (Algorithm 8). It may be noted that the column indexes corresponding to \(t_1, \ldots, t_l\) are stored in a list variable, named column and the corresponding masks are stored in a list variable, named column_mask. To check for the existence of the string in a file corresponding to, say, 1-th row of the index table, the second while loop of the algorithm unmarks the entries corresponding to \(I[i][\text{column}[1]], \ldots, I[i][\text{column}[l]]\) and checks if there exists a matching pair \((p_1, p_2)\), where \(p_1 \in I[i][\text{column}[1]]\) and \(p_2 \in I[i][\text{column}[2]]\) such that \(p_2 = MAc_k (p_1)\). This process is repeated for \((l-1)\) times for all \((\text{column}[i], \text{column}[i+1])\) pairs \(1 \leq i < l\) to identify the matching pairs \((p_1, p_{i+1})\) at each of the steps such that \(p_{i+1} = MAc_k (p_i)\). If \((l-1)\) steps are completed successfully, then that confirms the existence of the string in the file and the corresponding file identifier is added in the list named encrypted_file_pointers.

In this section, we described the algorithms for key generation, index generation, trapdoor generation and searching for our scheme \(\Pi_{sss}\). For better security, probabilistic trapdoor is desirable. Since the index is static, searching using probabilistic trapdoor and static index is difficult. Unlike previous schemes, our scheme produces probabilistic trapdoors and search algorithm accurately detects the string despite having static index. The correctness of this computation is explained in Remark 3. In the next section we provide the security of our scheme \(\Pi_{sss}\).

V. Security Analysis

Although few SSE schemes are available for string search, but most of them lack formal security proof against the revised definitions of [12] and also leak lots of informations beyond what is leaked from history [12]. In Subsection V-A we provide a formal proof of non-adaptive security of our scheme against honest-but-curious server. In Subsection V-B we propose modifications of our scheme to protect against active adversaries at the cost of more rounds of communications and memory space.

A. Honest but curious server

An honest-but-curious server follows the protocol and takes no actions beyond those of an honest server, and attempts to learn about the plaintext of documents or terms that were queried. The idea of non-adaptive security for SSE scheme for an honest-but-curious server was first introduced in [12]. In order to explain the non-adaptive security, we first provide the definition of history and trace.

Definition 3. (history) [12] Let \(\Delta\) be a dictionary and \(D \subseteq \mathbb{P}(\Delta)\) be a document collection over \(\Delta\). A q-query history over \(\Delta\) is a tuple \(H = (D, w)\) that includes the document collection \(D\) and a vector of q keywords \(w = (w_1, w_2, \ldots, w_q)\).

The access pattern induced by a q-query history \(H = (D, w)\) is given by \(\alpha(H) = (D(w_1), \ldots, D(w_q))\) [12]. The search pattern corresponding to q-query history \(H\) is a \(q \times q\) binary matrix \(\sigma(H) = (h_{i,j})\) such that \(h_{i,j} = 1\) if \(w_i = w_j\) [12].

Definition 4. (trace) [12] Let \(\Delta\) be a dictionary and \(D \subseteq \mathbb{P}(\Delta)\) be a document collection over \(\Delta\). The trace induced by a q-query history \(H = (D, w)\) is a sequence \(\tau(H) = ([D_1], [D_2], \ldots, [D_n], \alpha(H), \sigma(H))\) comprised of the lengths of the documents in \(D\).

We introduce the notion of history-of-string by extending the definition of history for string which is crucial for our security proof.

Definition 5. (history-of-string) Let \(\Delta\) be a dictionary and \(D \subseteq \mathbb{P}(\Delta)\) be a document collection over \(\Delta\). A q-query history over \(D\) is a tuple \(H = (D, s)\) that includes the document collection \(D\) and a vector of q strings \(s = (s_1, s_2, \ldots, s_q)\).

Let for the vector of q strings \(s = (s_1, \ldots, s_q)\), there are only \(q'\) distinct words, i.e., \(|\delta(s)| = q'\) and let these distinct words be \(w_1, \ldots, w_{q'}\). The access pattern induced by a q-query history-of-string \(H = (D, s)\) is given by \(\alpha(H) = (D(w_1), \ldots, D(w_{q'}))\) [12]. The search pattern corresponding to such a q-query history-of-string \(H\) is a \(q' \times q'\) binary matrix \(\sigma(H) = (h_{i,j})\) such that \(h_{i,j} = 1\) if \(w_i = w_j\) [12]. Similar to trace, we define trace-of-history-of-strings as follows:

Definition 6. (trace-of-history-of-strings) Let \(\Delta\) be a dictionary and \(D \subseteq \mathbb{P}(\Delta)\) be a document collection over \(\Delta\). The trace induced by a q-query history-of-strings \(H = (D, s)\) is a sequence \(\tau(H) = ([D_1], [D_2], \ldots, [D_n], \alpha(H), \sigma(H))\) comprised of the lengths of the documents in \(D\).

Lemma 1. Let for the vector of q strings \(s = (s_0, \ldots, s_q)\), there are only \(q'\) distinct words, i.e., \(|\delta(s)| = q'\) and let these distinct words be \(w_{0,1}, \ldots, w_{0,q'}\). Let \(H_0 = (D_0, w_{0,1}, \ldots, w_{0,q'})\) and also let \(H_0 = (D_0, w_{0,1}, \ldots, w_{0,q'})\). Similarly define \(H_1 = (D_1, s_1, \ldots, s_1, q)\) and \(H_1 = (D_1, s_1, \ldots, w_{1,q'})\). It is easy to check that \(\tau(H_0) = \tau(H_1)\) implies \(\tau(H_0) = \tau(H_1)\).

Now we provide the definition of non-adaptive indistinguishability security and non-adaptive semantic security for SSE from [12] with slight modifications for strings.

Definition 7. (Non-Adaptive Indistinguishability Security for SSE) Let \(SSE = (Gen, Enc, Trpdr, Search, Dec)\) be an index based SSE scheme over dictionary \(\Delta\), \(\lambda\) being the security parameter, and \(A = (A_0, A_1)\) be a non-uniform adversary. Consider the
in the definition of non-adaptive indistinguishability, there are only Remark 5.

Definition 8. \(\SSE\) is said to be semantically secure if for all polynomial size adversaries \(A\), the output of \(\Real_{\SSE, A}(\lambda)\) and \(\Sim_{\SSE, A, S}(\lambda)\) are indistinguishable. Let the simulator \(S\) on input \(\tau(H)\) generates \(v' = (\SST', c', t') = ((\Gamma', I', r'), (t'_1, ..., t'_q), (c'_1, ..., c'_q))\) as follows:

1. (simulating \(\SST'\))
   a. If \(q > 0\), set \(|\delta[H]| = q\). Choose uniformly at random the keys \(k'_r\) and \(k'_s\). From the trace, find all distinct words that occur in \(q\) strings. Let this collection be \(\{w_1, ..., w_{q'}\}\). Generate a \(n \times q'\) array \(I'\). Assign \(I'_r[i] = MAC_{k'_r}(w_i)^{-1}\) mod \(p\). Also compute \(I'_c[j] = Enc_{k'_c}(id(D_j))\) for \(j = 1\) to \(n\).
   b. For each string \(s_j = (w_{j,1}, ..., w_{j,k})\) and document \(D_i\) containing the string, find the masked hash-chain \((r_1, ..., r_l)\) as mentioned in Section IV.
   c. For each \(I'[i][j]\) such that \(|I'[i][j]| < \max\), run \(MAC_{k'_c}(...) \max - |I'[i][j]|\) many times on different random values from \(\mathbb{Z}_p\) and store the outputs at \(I'[i][j]\).
   d. If \(q = 0\), we allocate \(n \times \max\) (see Remark 6) array for \(I\) and fill up the corresponding cells with \(\max\) number of different random values from \(\mathbb{Z}_p\).

2. (simulating \(t'\))
   (\(t'_1, ..., t'_q\)) Compute \(t'_i\) corresponding to the string \(s_i = (w_{i,1}, ..., w_{i,k})\) as \(t'_i = (t'_{i,1}, ..., t'_{i,k})\), where \(t'_{i,j} = (t'_{i,j,1}, t'_{i,j,2}, ..., t'_{i,j,l})\) corresponding to the word \(w_{j}\), is computed as follows:
   \[ e = Enc_{k'_c}(w_j), \text{msk} = MAC_{k'_c}(w_j), \]
   \[ ci = MAC_{k'_c}(e + \text{msk} + ci), \]
   \[ t'_{i,j,1} = MAC_{k'_c}(e + \text{msk} + ci), \]
   \[ t'_{i,j,2} = e + ci, \text{and} t'_{i,j,3} = e + \text{msk}. \]

3. (simulating \(c'\))
   Set \(c'_i\) to \(\delta[D_i]\) if string chosen uniformly at random.

Now we show that each element of \(v'\) is computationally indistinguishable from its corresponding element of \(v\) to a distinguisher \(D\) that is given \(\SST\).

1. (\(\SST\) and \(\SST'\)) \(I'[i] = MAC_{k'_r}(w_i)^{-1}\) mod \(p\). Since \(MAC_{k'_r}(w_i)\) and \(MAC_{k'_c}(w_i)\) are indistinguishable, so is \(I'_r\) and \(I_r\). As with all but negligible probability, \(\SST\) does not include the key \(k_m\) and \(\pi\), the encryption scheme, is IND-CPA secure, this guarantees that each element of \(I_r\) is indistinguishable from its counterpart part in \(I'_r\). Due to the pseudo randomness of \(MAC()\) and all but negligible probability, \(\SST\) does not include the key \(k_m\), we argue that each of the \(\max\) number
of entries in $I[i][j]$ is indistinguishable from its counterpart in
$I'[i][j].$

2. $(t \text{ and } t^*)$ Since computation of $t$ involves computation using
$MAC()$ and $Enc()$ and with all but negligible probability, $ST_A$
does not include the key $(k_m, k_e, k_s)$ so from the pseudo-
randomness of $MAC()$ and IND-CPA security of $\pi$, $t$ and $t^*$
are indistinguishable.

3. $(c \text{ and } c^*)$ As with all but negligible probability, $ST_A$
uses the key $(k_s, k_m)$ and $\pi$ is IND-CPA secure, this
guarantees that each element $c$ is indistinguishable from its
counter part in $c^*.$

The non-adaptive indistinguishability guarantees the privacy of the
keyword from the index. However, it does not guarantee any security
against leakage of keyword from trapdoor. Towards this we introduce
the notion of search pattern privacy.

**Definition 9. (Search Pattern Indistinguishability)** Let $SSE$
be a dictionary based SSE scheme over dictionary $\Delta,$ $\lambda$
being the security parameter, and $A$ be a non-
uniform adversary. Consider the probabilistic experiment:

$$SPI_{SSE, A}(\lambda) :$$

- $K \leftarrow Gen(\lambda^k); (ST_A, H) \leftarrow A(\lambda^k)$
- Parse $H$ as $(D, s)$
- $(S, c) \leftarrow Enc_K(D)$
- let $w = (w_1, \ldots, w_q)$
- for $1 \leq i \leq q$
- $t_{b_i} \leftarrow Trpdr(w_{b_i})$
- $t \leftarrow (t_1, \ldots, t_{b_2})$
- $w_0, w_1 \leftarrow \triangle$ such that $|w_0| = |w_1|
- b \leftarrow \{0, 1\}$
- $t_{b} \leftarrow Trpdr(w_b)$
- $b^* \leftarrow A^\text{Trpdr}(\lambda^k); (t^*, K, t, w, t, H, ST_A)$
- if $b = b^*$ then return 1 else return 0

The advantage of $A$ in the above experiment is defined as
$Adv^{SPI}_{SPP, SSE, A}(\lambda) = \Pr[SP_{SPI_{SSE, A}}(\lambda) = 1] - \frac{1}{2}.$
SSE is said to be secure against search pattern indistinguishability
if for all polynomial size adversaries $A$, $Adv^{SPI}_{SPP, SSE, A}(\lambda) \leq negl(\lambda)$

**Theorem 3.** $\Pi_{SSE}$ is search pattern secure scheme against chosen
trapdoor attack in random oracle model.

**Proof:** We construct a simulator $S^{Enc, K}()$ that has the encryption
oracle for underlying private key encryption which is assumed to
be IND-CPA secure under the Definition 2. The simulator $S$
interacts with the adversary $A$ as follows:

1. (simulating $msk^*$ and $ci^*$) : Whenever $A$ queries for trapdoor
corresponding to some word, say $w_i$, $S$ maintains two lists,
namely $list_1 = (w_i, h^1_i)$ and $list_2 = (w_i, h^2_i)$ for $msk$
and $ci$ respectively which are initially empty. When $A$ queries for
trapdoor of $w$, $S$ responds as follows:
- if $w_i$ is in list_1 then it sets $msk = h^1_i$; else it chooses $h^1_i \leftarrow
\{0, 1\}. It makes an entry $(w_i, h^1_i)$ in list_1 and sets $msk^∗ = $ $h^1_i$. Similarly it simulates $ci^∗ = h^2_i$.

2. (simulating $t_{b_1}, t_{b_2}, t_{b_3}$) : $S$ asks the actual challenger to com-
pute $e = Enc_{K_m}(w)$ (to $S$ this is the encryption oracle). It then
performs the following computations:
- $t_{b_1} = e \oplus c^∗$; $t_{b_3} = e \oplus msk^*$; $t_{b_2} = (t_{b_1}, t_{b_2}, t_{b_3})$;
- It sets $t_{b_1} = h$ where $h$ is an entry corresponding to $e \oplus msk^∗ \oplus c^*$
in $list_1.$
to these columns of \( I \) that were being searched during the search by maintaining a book-keeping of columns being selected after every search.

There can be two cases. Firstly, server may manage to insert \( D_s \) into clients document collections \( D \) before the client’s processing of \( D \) to generate \((I \delta, c)\). To address such cases of spam detection is beyond the scope of this work and may be dealt by enabling spam filter. In the second case the client prepares and uploads \((I \delta, c)\) to a server depending on \( D \) which does not contain spam document. Client updates the document collections \( D \) and corresponding \((I \delta, c)\) whenever new documents are added or deleted. Server, after some time, manages to inject the spam document \( D_s \) without the knowledge of client. In this case we propose how \( \Pi_{ss} \) can be modified to resist such chosen document attack but at the cost of increased rounds of search.

**Modification of \( I \)**: Let initially there were \( n \) documents in \( D \) and \( D = \{D_1, \ldots, D_n\} \) and also let \( \delta(D) = \{w_1, w_2, \ldots, w_d\} \). Let after a modification of \( D \) now there are \( n' = n + 1 \) number of documents and the newly added document is a spam document. We denote this modified document collection by \( D' \). Let \( W = \delta(D') \cap \delta(D) \) be the set of common words. It can be observed that if the search query string consists words that are from \( \delta(D) \setminus W \), then access pattern will not involve newly added document. Also if the search string involves words that are only from \( D' \setminus W \), access pattern does not involve previous documents. An access pattern involving new and old documents is possible if and only if all the words in the search string are from \( W \). Note that client can always compute \( W \) if it maintains list of words say \( CI \). For each of the words in \( w \in W \) client treats these words as \( w[1] \) to distinguish them from \( w \) which already exists in \( CI \). Next, client adds all words of the form \( w[l] \) and words from \( \delta(D') \setminus W \) in \( CI \) and adds new columns in \( I \) for all newly added words. Client adds a new row in \( I \) for the newly added document and computes entries in each cell of the newly added row same as before.

**Two round communication for search**: To search for a string \((w_1, \ldots, w_l)\), such that all \( w_i \)'s are from \( W \), client prepares another search string namely \((w'_1, \ldots, w'_l)\) and performs two searches for \((w_1, \ldots, w_l)\) and \((w'_1, \ldots, w'_l)\), one by one. With this approach the access pattern does not mix old documents with the newly added one.

### VI. Performance Analysis

In this section we provide performance results of our proposed scheme \( \Pi_{ss} \). First we study the theoretical search time and index size needed in Lemma [2] and Lemma [3] respectively.

**Lemma 2.** Using \( \Pi_{ss} \), the search time for a query of \( l \)-word string is \( O(|\delta(D)| \times l \times n) \) operations in \( Z_p \), where the underlying document collection is \( D \) having \( n \) documents.

**Proof:** Recall that in the index table there are \( |\delta(D)| \) number of columns corresponding to \( |\delta(D)| \) distinct keywords. Since \( \Pi_{ss} \) uses non-deterministic trapdoor, searching for column corresponding to each keyword takes \( O(|\delta(D)|) \) number of operations in \( Z_p \). After \( l \) columns are found, for each document, \( MAC() \) may be run to check for the adjacency of words \( l \) times, each of which takes constant time \( O(1) \). Hence the result follows.

**Lemma 3.** The index size needed for implementing \( \Pi_{ss} \) is \( O(n \times |\delta(D)| \times f \times \lambda) \).

**Proof:** Index is a \( n \times |\delta(D)| \) array where each cell contains \( f \) number of entries from \( Z_p \), each being \( \lambda \) bit string as \( p \) is a \( \lambda \) bit prime.

**Will parallelization improve the performance?** We note that, by the construction of index, a row of the index, which corresponds to a document in the document collection, is independent of other rows and thus processing with one row has no overlap with that involving other rows. This crucial property enables the searching algorithm (Algorithm 2) conformance for parallelization. Currently the searching algorithm is linear in \( n \), the number of documents (see Lemma 2). On parallelization of searching into \( t \) number of threads, each assigned with \( n' = \frac{n}{t} \) rows, the searching will be linear in \( n' \) and since each of these threads will run in parallel, from Lemma 2 the complexity will be \( O(|\delta(D)| \times l \times n') \) resulting into huge performance gain, specially in high load environments.

Now in the following subsections, we provide comparison of our scheme with schemes in [18] and [23] in terms of storage and server side computation in the setup of string search. It may be noted that unlike these schemes, we don’t need any storage of data owner. Moreover, unlike our scheme, none of these schemes are proven to be secure under the newly introduced definitions of non-adaptive security of [17].

#### A. Comparison with bloom-filter based scheme of [18]

In [18], authors proposed phrase search using bloom-filters. We start with converting their results related to complexity analysis in terms of our parameters for making it conformance for the comparison. In [18], authors used \( N, x, q, b_2, b_3 \) and \( k \) to denote number of documents, number of distinct keywords, number of words in the query, number of keyword pairs, number of keyword triplets and number of bloom filter hash functions respectively. Converting into our notation, we have \( N \approx n, x \approx |\delta(D)| \) and \( q \approx l \). Also \( b_2 = O(|\delta(D)|^2) \) and \( b_3 = O(|\delta(D)|^3) \).

**Cloud side computation.** In [18], authors showed that the computational cost of cloud server is \( \#c = Mod(k(q - 2)N) + And(Nb_3) \). Also from [18], to insert \( |\delta(D)| \) keywords in a \( m \)-bit bloom-filter, the minimum false-positive probability \( p \) is achieved when \( k = \frac{m}{|\delta(D)| \ln(2)} \) and \( m = 1 - \frac{|\delta(D)|^2}{(\ln(2))^2} \). Unlike the scheme of [18], in our scheme the probability of false positive rate is zero. So even when the scheme of [18] achieves minimum false positive rate, the cost of computation in asymptotic notation becomes

\[
\#c = Mod(k(q - 2)N) + And(Nb_3)
\]

\[
= O\left(\frac{m}{|\delta(D)|} \times l \times n\right) + O\left(n \times |\delta(D)|^3\right)
\]

\[
= O\left(n \times |\delta(D)|^3\right)
\]

Since \( |\delta(D)|^2 \gg l \), the scheme in [18] is more expensive in terms of cloud server computation.

**Cloud side storage.** In [18], authors showed that the storage at cloud server is \( \#s \), where,

\[
\#s = x \times (\log_2(x) + N) + N((b_2 + b_3)
\]

\[
= O\left(|\delta(D)| \times \log_2(|\delta(D)|) + n + n(|\delta(D)|^2 + |\delta(D)|^3)\right)
\]

\[
= O\left(|\delta(D)| \times \log_2(|\delta(D)|)\right) + O\left(n \times |\delta(D)|^3\right)
\]

\[
= O\left(n \times |\delta(D)|^3\right)
\]

We observe that in the setup of string search where we consider all words as key words, \( x \times f \ll |\delta(D)|^2 \). So the scheme in [18] incurs more storage in cloud compared to our scheme.

#### B. Comparison with TSet based scheme of [23]

In the TSet based approach of string search of [23], the computational cost of server side searching is exponential in \(|l| \times (m - 1)|\).
number of multiplication operations in $\mathbb{Z}_m$ (see [23]), where $|t|$ is the number of outputs of TSet and $m$ is the length of query and $\log_2(q) = \lambda$ is the security parameter. Also from the TSet construction, which incorporates the combinations of keywords, $|t|$ itself is exponential on the distinct keywords, which, in our notation, is $\delta(D)$. Thus the cost of sever side searching is exponential in $|\delta(D)|$ which is linear in our case (see Lemma 3).

C. Experiment on Speech Data

From Lemma 2 it is clear that search time is linear in number of documents. Now we validate our scheme against two different commercial datasets ([1], [2]). We build a symmetric search engine based on the scheme $\Pi_{\text{ss}}$ using Java in linux platform. The implementation is done on Asus A Series Core i3 laptop (4 GB/ 1 TB HDD) 90NB0652-M32310 XX2064D. We use Java based ‘Jpair’ library for the cryptographic primitives.

The TIMIT speech corpus [TIMIT] of speech was collected in 1993 as a speech data resource for acoustic phonetic studies and has been used extensively for the development and evaluation of ASR studies. TIMIT contains broadband recordings of 630 speakers of eight major dialects of American English, each reading ten phonetically rich sentences. The corpus includes time-aligned orthographic, phonetic and word transcriptions as well as a 16-bit 16 kHz speech waveform file for each utterance. TIMIT was designed to further acoustic-phonetic knowledge and automatic speech recognition (ASR) systems. It was commissioned by DARPA and worked on by many sites, including Texas Instruments (TI) and Massachusetts Institute of Technology (MIT), hence the corpus’ name.

Phone recognition in TIMIT has more than two decades of intense research behind it and its performance has naturally improved with time so that it can be useful for researchers, professionals and engineers specialized in speech processing when considering future research directions. TIMIT is the most accurately transcribed speech corpus in existence as it contains not only transcriptions of the text but also contains accurate timing of phones calls. This is impressive given that the average English speaker utters 14-15 phonic sounds (equivalent to phone symbols) a second. A more detailed description of the data may be obtained at [1].

D. Performance against Speech Dataset

The performances of string search on the speech data [1] are compared for two modes of implementations. The implementation of the scheme using masked hash-chain and padding using random values from $\mathbb{Z}_p$, $f$ number of times as mentioned in Section [IV] is referred to as ‘no leakage’. In another implementation we use masking over integers denoting word positions without padding. Here the frequency as well as the positions of the searched words in the documents are leaked, but after the search is done. So we refer to this as ‘partial leakage’.

Figure 1 shows a comparison of index generation time for these two methods. Figure 2 shows a comparison of search time for these two methods. Search time mainly depends on the number of documents through which the search is to be performed. So far we have performed search on a string of length 5. So Figure 2 shows how search time varies with respect to the number of documents. Unlike [13], in our implementation, search time depends not only on the number of documents returned, but also on the number of documents in which the keywords to be searched are present. This is due to the fact that for efficient implementation, we first check if all keywords are present in a document. If all key words are present in a document then and only then we check for the adjacency. It may be noted that the major share of search time is taken by checking adjacency. So search time may also increase in these cases where all keywords are present in many documents, but not in a consecutive fashion, yielding zero number of documents to be returned.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.png}
\caption{Index generation graph comparing index generation time using padded and masked hash chain technique with index generation time using position integers}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2.png}
\caption{Search time graph comparing the search time using hash chain technique with the search time using position integers}
\end{figure}

E. Experiment on DNA Data

We have implemented our algorithm for searching sequences of SNP’s (Single Nucleotide Polymorphism) in large genome databases which is available at [2]. SNP represents a difference in a nucleotide which is a single DNA building block. In the data set there are 10000 binary DNA sequences, each of which is 2185-bit long. We assume that the bit pattern to be searched is of length at least 8 and encode the binary sequences by using a set, $\mathcal{K}$, of $2^8$ i.e. 256 distinct symbols corresponding to every 8-bit of the data starting from the beginning. We treat each of these encoded sequences over $\mathcal{K}$ as different strings over $\mathcal{K}$. Search time in a document depends on the depth of the strings and the number of sequences. Since these sequences are very long compared to the speech data of Subsection VI-C time taken for searching over this dataset is more compared to the time taken for
searching over speech data. We apply our scheme \(\pi_{pas}\) on this data to search a string of length 3 in the encrypted domain, which is a same as searching 24-bit binary string in the binary sequence data. Due to huge size of the DNA data, in the implementation, we use directly integer pointers without masking and hash-chaining as otherwise the size of index is becoming larger than the available memory.

F. Performance against DNA Dataset

Figure 3 illustrates the time needed to generate index table for the DNA sequence, where we plot the time (in seconds along Y-axis) needed against number of encrypted DNA’s (along X-axis). This graph also reflects a linear growth with the increase in number of DNA sequences. Figure 4 compares the performance of searching patterns over encrypted DNA sequences of length four over the set \(\mathcal{K}\). So in binary it is a string of length 24. Here also we plotted time of search in seconds along Y-axis against the number of DNA sequences to be matched along X-axis. This graph also shows roughly a linear growth with the increase of number of encrypted sequences.

VII. CONCLUSION

With the increasing number of documents stored in cloud, searching for the desired document can be a difficult and resource intensive task. One solution may be to use symmetric searchable encryption (SSE) which allows one party to outsource the storage of its data to another party (a cloud) privately while enabling to search selectively over it. In this paper we revisited the security definitions of [12] and proposed a new lightweight SSE scheme \(\Pi_{pas}\) for string search. We have shown that our scheme is secure under the non-adaptive indistinguishability definition [12]. For active adversary, we propose modification of the scheme \(\Pi_{pas}\) at the additional cost of memory at client’s end and two rounds of communications for one modification of document collection. Towards this direction, future research can be performed to design efficient SSE scheme ideally with one round of communication. With our scheme, server does not learn the information related to word frequency and word positions except what it can learn from the history.

We, for the first time, introduce new security notion in SSE, named, search pattern indistinguishability. It may be observed that with non-adaptive indistinguishability security, although the keywords are guaranteed to be secure from the possible leakage from index, however it does not guarantee the security from the possible leakage from trapdoor. Towards this, we for the first time introduce probabilistic trapdoor and prove that our scheme is secure under such criterion. We have implemented our scheme for the first time to search over phone symbols and validated it using the TIMIT dataset. We have also implemented our scheme over DNA data of [2] and successfully achieve pattern matching functionality over encrypted domain.

While dealing with string search, designing a SSE scheme satisfying adaptive-indistinguishability-security definition of [12] seems intuitively impossible. This is because to generate an index in advance which is consistent with future search, one unavoidable assumption needed is the presence of all possible strings in each document. This can be done by considering all permutations of keywords for every document which makes the index size exponential in \(n\) for \(n\)-document collection. According to the definition of [12], index size is linear in \(n\) which is essential from efficiency point of view. From the angle of this intuition, future research can be carried out to give a formal proof in support of non-existence of adaptively secure SSE scheme for string search. In this paper we have considered honest-but-curious adversaries and active adversaries. Also, designing SSE scheme for string search with adaptive-indistinguishability-security against some newly defined adversary can be a future research direction.

ACKNOWLEDGMENT

This research was partly supported by the Engineering and Physical Sciences Research (EPSRC) UK project “Trusted Things & Communities: Understanding & Enabling A Trusted IoT Ecosystem” and European Commissions Horizon 2020 research and innovation project RED - Alert under grant agreement No 740688.

REFERENCES


