Storage Efficient Substring Searchable Symmetric Encryption

Iraklis Leontiadis
Ecole Polytechnique Federale de Lausanne (EPFL)
School of Computer and Communication Sciences
iraklis.leontiadis@epfl.ch

Ming Li
University of Arizona
Department of Electrical and Computer Engineering
lim@email.arizona.edu

ABSTRACT
We address the problem of substring searchable encryption. A single
user produces a big stream of data and later on wants to learn the
positions in the string that some patterns occur. Although current
techniques exploit auxiliary data structures to achieve efficient
substring search on the server side, the cost at the user side may be
prohibitive. We revisit the work of substring searchable encryption
in order to reduce the storage cost of auxiliary data structures. Our
solution entails a suffix array based index design, which allows
optimal storage cost $O(n)$ with small hidden factor at the size of
the string $n$. Moreover, we implemented our scheme and the state
of the art protocol [7] to demonstrate the performance advantage
of our solution with precise benchmark results.

CCS CONCEPTS
• Security and privacy → Database and storage security; Security protocols;

KEYWORDS
substring searchable encryption, suffix array, secure outsourced
storage, compression index, privacy

ACM Reference format:
Iraklis Leontiadis and Ming Li. 2018. Storage Efficient Substring Searchable
Symmetric Encryption. In Proceedings of 6th International Workshop on
Security in Cloud Computing, Incheon, Republic of Korea, June 4, 2018 (SCC’18),
15 pages.
https://doi.org/10.1145/3201595.3201598

1 INTRODUCTION
Nowadays, there is a flourish of protocols delegated to run by an
untrusted coalition of servers, systems, services, called hereafter
the cloud. Due to the untrusted nature of the cloud, users seek to
protect the privacy and security of their data with cryptographic
primitives. The cloud on the other hand offers an economy of scale
with the impressive resources it acquires, ranging from software
to hardware. Usually users need to perform a search on their data.
Tailored protocols for secure searchable encryption have been pro-
posed in the literature, whereby single or multiple users upload
encrypted documents, with some auxiliary data structure called an
index, allowing the cloud to correctly return documents contain-
ing a single, multiple or a boolean function of keywords, without
compromising index, query, and documents privacy.

While keyword based search protocols are quite common in a
large range of applications, they cannot efficiently address all the
possible queries a user submits to the cloud. Substring based queries
have come to the forefront due to the ubiquitousness of devices and
the progress in storage technology. Devices produce a big stream
of data, which needs to be queried later on with substring based
queries. Namely a substring query for a stream of data, consists of
a substring of the stream and the result is the position of the substring
in the big stream, or/and the number of occurrences of multiple substrings.

Applications. In a health-care application, data enclaves which
hold giant stream of medical information such as DNA sequencing
are asked to answer substring queries by medical labs. The possible
position of a substring in the whole DNA sequence of a single
person gives information about predisposition to diseases. As such,
it is treated as personal sensitive information and should be pro-
tected. Nowadays, the sequencing process is possible thanks to the
progress of computers. Online services offer DNA sequencing to
institutions and individuals. In the logging systems scenario, com-
panies, institutions and organizations produce log data of giant size.
The logs are recorded and uploaded in a cloud infrastructure to take
advantage of the cheap storage space. Log data are often searched to
identify malicious substring patterns. The position of the suspicious
searched string token will act as a bookmark to further download
the logs data, which proceed and succeed that position for further
investigation. Deep packet inspection (DPI) is another application
whereby a gateway, firewall, or Intrusion Detection System (IDS)
on behalf of a user is looking for prohibitive content on a bigger
stream. In general, the vast amount of information renders substring
queries a real challenge and reducing the storage cost of the
encrypted index would increase the performance of such services.

Protecting the privacy of the data stream and the substring query,
while allowing an untrusted cloud to correctly answer substring
matching pattern efficiently and securely is not trivial. Following
the searchable encryption approach, separating the data itself from
the index, results in a prohibitive storage index cost $O(n^2)$, where $n$
is the size of the stream. The index consists of all possible substrings
of a stream of data of size $n$ and the encrypted data are the positions
of the substring. Recently, the authors in [7] proposed a solution that
asymptotically achieves $O(n)$ storage costs by exploiting
the auxiliary data structure of the suffix tree. However the asymptotic
costs of $O(n)$ hide a constant factor that can be roughly up to $20$
[1, 3, 24] for the construction of the suffix tree due to the complexity
of the tree and the extra pointers to traverse a tree. Moreover
the suffix tree based approach leaks unnecessary information that
eventually can reveal all the encrypted positions of the substrings. Following a different approach other than auxiliary index based methods, the authors in [12] achieve to hide the extra leakage at the cost of fixed length substring patterns. The neat of their solution lies on the design of subset sum problems tailored to the positions of specific substrings, so as to the cloud can solve it partially. However this comes at the cost of small constant substring query length during the execution of the protocol.

Our goal, which launches our research is to reduce the increased size of index for substring queries, which has to be computed once and be kept at the cloud during the entire lifetime of the protocol. Even a small improvement would have big impact as the data to be indexed for substring queries span to million of elements. Moreover, besides the cloud side cost cuts, the client will also be positively affected as the smaller the size of the index it is outsourced, the less the charges clients commit to the cloud.

Idea. After encrypting the suffix tree of Chase scheme [7], the encrypted structure leaks a lot of information concerning the internal structure of the tree as number of leaves, children and double “touched” branches. The authors suggested a dummy node policy in order to hide as much information as possible. After constructing the suffix tree with N nodes, the suffix tree is filled up with 2n – N internal nodes. To each node with less than σ children, where σ is the size of the vocabulary, up to σ dummy nodes are appended. Encrypting all these dummy blocks drastically increases the storage overhead and subsequently the communication cost of the protocol for index construction.

Our core idea lies at the properties of a suffix array based indexing. A suffix array contains information about the position of each suffix of a string and has constant size n for a string of n size. In contrast, the data structure of the suffix tree has no constant size and can acquire up to 2n nodes, with each node storing information about its edges, parent and children nodes, thus increasing the storage need. By choosing the suffix array we decrease the storage need for the construction of the index. The second factor, which allows for less storage and subsequently communication efficiency is the dummy blocks policy which is used to hide the structure of the suffix tree in [7]. Our dummy node policy to obscure the encrypted suffix array relies only on the frequency of the most frequent character. Namely, we fill up the original string with characters such that the frequency of each character is the same. However this approach raises a shortcoming when applied to data sets with skewed frequency distributions such as text based datasets. We overcome that limitation with a bucketization technique. Instead of building the index on a single character approach we explore the idea of grouping together characters consisting a bucket. Surprisingly our experimental evaluation showed that the skewed frequencies are diminished and the final storage overhead for the index is considerably smaller by a factor of 1.8 for text based datasets.

In this paper we design and analyze a storage efficient Substring Searchable Symmetric Encryption (S²E) protocol with variable size of substrings. We follow a different approach from existing techniques that allows us to achieve the efficiency, functional and security goals we want. In our technique we exploit a self-indexed data structure, which allows the cloud to search for substring queries. Its form resembles the suffix arrays with some additional extra steps. The main contributions in the paper are summarized as follows:

- **Storage efficient Substring Searchable Symmetric Encryption (S²E):** Thanks to the employment of the suffix array, which achieves a small hidden factor (∼4) in the \(O(n)\) asymptotic complexity, compared to the bigger (∼20) hidden factor of the suffix tree, our design presents a storage efficient searchable substring symmetric encryption protocol.
- **Variable substring query length:** Our solution allows a dynamic issue of substring queries of variable size without the need of defining a fixed query size beforehand.
- **provably secure.** Our scheme is provably secure in the real-ideal simulation paradigm with similar leakages as the state of the art scheme in [7].
- **Prototype implementation:** We implemented our protocol and the state of the art work in [7]. We performed a real world comparison of both schemes based on computation time and communication overhead to build the index and query for a substring. Our results show a performance advantage of 1.8 storage overhead on text based datasets as the enron email one.

Outline. In section 2 we introduce the problem this paper addresses. Afterwards, in section 3 we review similar cryptographic protocols for substring searchable symmetric encryption. We continue in section 4 with preliminaries of our solution and the basic building blocks. In section 5 we illustrate in more details the design components of our protocol and in section 6 we describe in details our protocol. We present a thorough security analysis in Theorem 7. In section 8 we present some benchmark results and in section 9 a comparison with state of the art. Finally, we conclude in section 10.

# 2 PROBLEM STATEMENT

In this section we formalize the problem of string matching. We first start with the functional requirements of substring matching and afterwards we present the security requirements of the protocol.

## 2.1 Functional Requirements

Herewith pattern matching, string matching and substring matching are used interchangeably in this paper. We assume that a string \(S\) is modeled as a one dimension array \(S[1...n]\). A substring is another array \(T[1...m]\). The elements of each array are drawn from some finite alphanumerical alphabet \(\Sigma\) of size \(|\Sigma| = \sigma\). We say that a substring \(T\) occurs in \(S\) if there exists \(s : 1 \leq s \leq n - m + 1\) and \(S[s + 1...s + m] = T[1...m]\), meaning that \(S[s + j] = T[j], 1 \leq j \leq m\).

Naïve algorithms for pattern matching achieve \(O(n)\) on search time and \(O(1)\) cost on preprocessing. The algorithm simply scans all the positions \(i, 1 \leq i \leq n - m\) of the string \(S\) until it finds \(m\) consecutive matches at a position \(j, 1 \leq j \leq n - m + 1\). Trading preprocessing efficiency for better search costs, Robin Karp algorithm [21] achieves \(O(n - m + 1)\) search time and \(\Theta(m)\) preprocessing amortized cost. In a similar trajectory Knuth-Morris-Pratt [22] has \(\Theta(n)\) search complexity and \(\Theta(m)\) preprocessing time. Boyer-Moore pattern matching technique [2] increases the preprocessing cost at \(\Theta(m + \sigma)\) in order to have worst case search complexity \(O(n)\). Following a different trajectory substring matching techniques achieve
O(m) search time by leveraging a more sophisticated preprocessing
step, in which the suffixes of all substrings are computed along with
their positions in the string S, be it suffix tree [25, 34, 35] or suffix
array [24]. Suffix tree though has a more expensive space efficiency
due to the extra information the suffix tree has to keep [1, 3, 17, 24].
This cost is translated to a constant factor that approximates
≈ 20, which is hidden in the O(n) asymptotic storage cost of the suffix
tree construction. As a first step to relax this storage extra hidden
cost we choose to build upon the suffix array string matching ap-
proach which has a much simpler storage cost which approximates
4n [1].

We redraw upon the queryable encryption syntactical definition of
[7], since we believe it follows a deceptive abstraction. Namely,
the functional definition claims to capture a generic framework
for searchable encryption, in the sense that a query F can be any
function keyword query, or substring query. However, an encrypted
searchable encryption scheme is a more generic protocol, since it
can be used to solve the substring searchable encryption problem
with the encrypted inverted index technique as shown in the intro-
duction. As such, searchable and substring encryption schemes
cannot be addressed by the same definitional framework. Furthermore,
the nature of the problem and the solution for substring queries
drastically varies from keyword searchable encryption, since the
index contains the data and there are not two separate objects,
meaning that the index for substring queries is self-indexed, since
from the index you can recover the underlying data structure. In
contrast in encrypted searchable encryption, there is a clear distinc-
tion between the index, and the data structure that holds the data
/files with keywords). For these reasons we rewrite the functional
definition for substring searchable encryption.

**Definition 2.1.** A Substring Searchable Symmetric Encryption
scheme (S\(^SE\)) is a collection of four polynomial time algorithms
(KeyGen, PreProcess, SrchToken, Search) defined as follows:

- k ← KeyGen(1\(^λ\)): It is a probabilistic algorithm that takes
  as input the security parameter in the unary form 1\(^λ\) and
  outputs the secret substring search key k.
- SES ← PreProcess(k, S): This algorithm takes as inputs the
  string S and the secret key k and outputs the substring
  encrypted data structure SES.
- tk\(_T,S\) ← SrchToken(k, T[1...m]): It is a probabilistic al-
  gorithm that takes as input the secret substring search key k,
  a string T[1...m] and outputs a trapdoor to search for the
  string T on data stream S, through SES.
- (s,⊥)← Search(tk\(_T,S\), SES): It is a deterministic algorithm
  which takes as input a trapdoor tk\(_T,S\) and a substring en-
  crypted structure SES and outputs the positions s in S that
  substring T occurs, or ⊥ otherwise.

A substring searchable encryption scheme is correct if ∀k ∈ N, VS ∈ Σ, tk ← KeyGen(1\(^λ\)), ySES ← PreProcess(k, S), tk\(_T,S\) ← SrchToken(k, T[1...m]), Search(tk\(_T,S\), SES) always returns the cor-
rect positions s in the string S or ⊥ otherwise.

2.2 Security Model

Intuitively the security guarantee we ask for is 1) given a prob-
abilistic polynomial time adversary A with access to a sub-
string encrypted structure SES, A cannot gain more partial
information about the underlying stream of data S and 2) given
a set of trapdoor tokens for an adaptively generated set of queries
q = (q\(_1,q_2,q_3,\ldots,q_o\)) associated with set of tokens
t = (tk\(_1,tk_2,tk_3,\ldots,t_k\)) A cannot learn anything for q and t.
Following the symmetric searchable encryption paradigm we know
it is impossible to achieve those two security guarantees without
leaking some extra information as the observed in [5, 6, 8].

We express the security guarantees of the protocol in terms of simu-
latability [23]. First a leakage function L is defined, which ex-
presses the leakage of a S\(^SE\) scheme to an adversary A, through
the transcripts of the protocol. The simulation framework assumes
two worlds. The Real\(_{A(S(λ))}\) world, whereby adversaries can corrupt
the parties they want and the Ideal\(_{A,S(λ)}\) one in which there is
only benign behavior of each party. The security analysis narrows
down to the design of a simulator S, who tries to simulate the ma-
licious behavior in the Ideal\(_{A,S(λ)}\) world, only through access to
the leakage function L. We say that a protocol is secure if S simu-
lates indistinguishable views of the adversary A in the Ideal\(_{A,S(λ)}\)
world.

The adversary A plays the role of a semi-honest cloud and during
the two world we assume a challenger C who interacts with A. We
describe the two world in algorithmic details in what is follows:

- **Real\(_{A(S(λ))}\) world:**
  - C runs KeyGen(1\(^λ\)) to obtain k.
  - A chooses a string S ∈ Σ, sends it to C and C replies with
    SES ← PreProcess(k, S) to A.
  - A issues a polynomial number of adaptively chosen queries
    q = (q\(_1,q_2,q_3,\ldots,q_o\)) and receives from C a set of tokens
t = (tk\(_1,tk_2,tk_3,\ldots,t_k\)).
  - Finally A outputs v = (SES, t).

- **Ideal\(_{A,S(λ)}\) world:**
  - A outputs a string stream S.
  - The simulator S through the leakage L generates SES and
    forwards it to A.
  - A issues a polynomial number of queries q =
    (q\(_1,q_2,q_3,\ldots,q_o\). S replies to each of the queries through
    the leakage function L with t = (tk\(_1,tk_2,tk_3,\ldots,t_k\)).
  - Finally A outputs v = (SES, t).

**Definition 2.2.** A substring searchable encryption scheme scheme
is adaptively L-semantically secure against a probabilistic poly-
nomial time adversary A if there is exists a polynomial Simulator S
such that for all polynomial time distinguishers D:

\[
|\Pr[D(v) = 1 : v ← Real_{A(S(λ))}] - \Pr[D(v) = 1 : v ← Ideal_{A,S(λ)}]| \leq neg(λ)
\]

3 RELATED WORK

Computing on encrypted data goes beyond linear and affine trans-
formations on numerical data held by an untrusted cloud. It is very
common for a single or multiple users to search on encrypted data remotely. Under the scenario of remotely searching, new security guarantees beyond confidentiality and integrity need to be considered. The so called search pattern reveals similarities between search queries, and the access pattern leaks the identifiers or memory addresses of the accessed files—even encrypted [8].

The ORAM paradigm [16] enables a user to remotely search for encrypted data, without leaking the search or the access pattern. The trade off comes with a bandwidth and communication burden. In [16] the bandwidth overhead is polylogarithmic, which has been reduced down to logarithmic in subsequent work [10, 27, 31–33]. However, in order to provide a real practical real world remote search protocol on encrypted data some leakages are allowed: the search and access pattern. The formalization of these patterns has been presented in the literature under the Symmetric Searchable Encryption (SSE) framework [4–6, 8, 29], with efficient instantiations.

With SSE a user encrypts data and index separately. It uploads both to an untrusted cloud and later on can search efficiently file identifiers with specific single keywords or an expressive boolean function over keywords, without the cloud learning anything about the files or the keywords. This comes at a security cost of leaking the search and access pattern. Following the approach of SSE, we can design substring searchable symmetric schemes as follows. The user builds an index which maps substrings to positions, encrypts the index and uploads it to the cloud. Later on, the user computes a token for the specific substring and the cloud tries to find a match in the index. If a match occurs the cloud returns the encrypted positions for this token, which correspond to a substring. However, this approach has increased storage cost \(O(n^2)\), since the cloud has to keep track of all the possible substrings.

Tailored substring searchable encryptions schemes have been proposed in the literature [7], [12], [11]. Chase et al. [7] leverage the auxiliary data structure of the suffix tree. A suffix tree is a compressed suffix trie, can be computed in time \(O(n)\) and allows for substring search in \(O(m)\) time on a substring of size \(m\). Its amortized storage cost \(O(n)\) hides a big constant factor, which could be equal to 20 [1, 3, 17, 24]. In [11] the authors extended the efficient SSE scheme for boolean queries from [4] in order to support substring matching. The idea is to build an index of overlapping k-grams, to preprend its relevant position and encrypt it. When a user needs to perform a substring query, the cloud performs a conjunctive keyword search for all the k-grams of the substring and returns the position. The disadvantage of the scheme comes at the need of storing all the overlapping k-grams at the cloud, which will represent substrings.

In [12] the authors follow a different approach. Instead of taking the index-then-encrypt approach with fast symmetric cryptographic primitives, they modify the subset sum problem, which is used to build public key encryption schemes, in a means such that the cloud can solve it. This technique hides also the search pattern but comes at the cost of fixed size substrings, that must be defined in the beginning of the protocol. Moreover the substring should be substantially small with respect to the big stream. Our solution in contrast allows variable size of substring of any size.

Recently, Blass and Moataz [26] strengthen the security requirements by hiding the search and access patterns, following the ORAM approach. By leveraging the Path ORAM technique and the suffix array construction for substring queries, the authors manage to reduce the bandwidth, with a binary recursive tree above the position map. Each node in the tree represents a Path ORAM of the binary search tree for the suffix array. However, in order the cloud to be able to perform an oblivious binary search has to keep track of all the suffixes, which blows up the storage cost for the server. Furthermore the need for storing the suffixes cancels out the suffix array storage advantage over suffix tree. Finally, due to the Path ORAM technique the user has to store a state logarithmic on the length of the string—for the position map. The extra security guarantees of the tailored ORAM scheme do not allow for efficient storage cost both at the client and the cloud side, which is the goal for our work. Sanders et al. [30] presented a public key substring searchable encryption, where the underlying substring algorithm is a variant of Rabin-Karp algorithm, thus suffering from linear substring search time. It is noteworthy though to mention that their goal is to decouple keyword generation from the encryption of data in order to allow searches even when the underlying text is not known when the index is built.

Papadopoulos et al. [28] addressed the problem of authenticating substring queries without privacy and various work for pattern matching adopts the two party computation model [9, 14, 18, 19] in which one party holds the data stream and a client the pattern. The model differs from the substring searchable symmetric encryption, since in the latter one client holds both the pattern and the stream and uploads an index of the stream to an untrusted party.

4 PRELIMINARIES

In order to reduce the storage cost for our Substring Searchable Symmetric encryption scheme (SSE) we first substitute the storage expensive suffix tree of the state of art work in [7] with a suffix array \(SA\). A suffix array for a string \(S\) of size \(n\) constitutes of an integer array of size \(n\), which has at each position a pointer to the start of the matched suffix \(T[1...m]\) in the string \(S\). \(SA\) is lexicographically sorted with respect to all the possible suffixes and can be computed in linear time on the size of the string \(S\). In order to look for the position of a substring, a binary search in \(SA\) is performed, which is used as an index to the original string. Thus, the running time for a substring search is \(O(m + \log n)\). Let us now consider a concrete example to uncover its details. Suppose \(S=\text{lalakis}\). The algorithm for the suffix array proceeds as follows:

1. Compute all the suffixes starting from the right-most position: \(s, is, kis, akis, alakis, lalakis\).
2. Lexicographically sort the suffixes: \(akis, alakis, is, kis, lakis, lalakis, s\).
3. Find the position in \(S\) of each suffix from step 2 and store them in an array \(SA = [4, 2, 6, 5, 3, 1]\)
4. Output \(SA\).

However, plugging the \(SA\) for a substring searchable symmetric encryption scheme raises some difficulties. We assume that the suffix array is encrypted under a secret key of the user. In order to retrieve the correct encrypted index position from \(SA\), the cloud
should run a binary search obliviously without learning the underlying string \( S \), query substring \( T \), or any of the suffixes. A solution to the problem is to use the technique presented by Gentry et al. [15], which allows for a single ORAM query in order to perform a binary search over encrypted data. However, in order to adapt this approach it is required from the server apart from the encrypted suffix array, to store the tree of the encrypted data, which would be an extra burden for its storage complexity.

We take advantage of the self-indexed data structure Ferragina-Manzini index, called hereafter FM index [13]. Namely, from FM index the untrusted cloud can answer substring queries by leveraging the suffix array \( SA \), without the need for an ORAM query. The neat property of the FM index is that it can reconstruct the original string \( S \) with some extra auxiliary data structures, thanks to its instantiation from the Burrows-Wheeler Transformation algorithm (BWT) [3]. For the reconstruction it employs the LF mapping technique, thus there is no need to store the encrypted stream \( S \). The FM index can be derived from \( SA \), as such its computational overhead is almost for free, after the computation of the suffix array. We describe the core building blocks of the FM index in what it follows.

### 4.1 Pattern matching

In this section we describe the compressed index FM, that will be used for the construction of our Secure Pattern Matching (S’E) protocol. The design lies heavily on the BWT transformation for compression of bit-strings and on a special LF mapping for the reconstruction of the original string from BWT. The BWT, along with the LF mapping technique and some auxiliary information are the basic blocks of the compressed index for substring queries.

#### 4.1.1 BWT Transformation

The Burrows-Wheeler Transformation (BWT) transforms a stream of data by leveraging the entropy of each character. In a nutshell, the data stream \( S \) is transformed to an encoding \( W \) such that compression algorithms provide high rate of compression. For ease of completeness we show the steps to transform an original stream \( S \) to \( W \) with BWT in algorithm 1. First, the algorithm appends the terminating symbol \( \$ \) to the input string \( S \). Then, it builds the matrix \( W \) by permuting the symbol \$ . At each iteration the permutation is appended as a new row to the matrix \( W \). Finally the rows of \( W \) are sorted lexicographically in an ascending way.

#### 4.1.2 LF Mapping

The LF Mapping technique takes the first \( F \) and last \( L \) columns from the BWT transformation and through an iterative process (algorithm 2) reconstructs the original string \( S \). Starting from the first elements of each column from \( F \) and \( L \), the algorithm employees \( L \) as an index to the \( F \) column. Each time the element of the \( L \) column is appended to a LIFO stack. The value at the current position will be used as an index for the \( F \) column for the next loop. An example is presented in figure 1. At the first iteration the pointer indicates the first position in both columns \( F \) and \( L \). For the next iteration the \( L \) character ‘\$ ’ indicates the index for the first column \( F \), which can be found at its last position with \( F[7]=\$ \). The current character at the \( L \) column is appended to a stack \( D \). For the next iteration the current character at the \( L \) column indicates the next index for the \( F \) column. The character \( i \) is pushed to the stack \( D \). The procedure halts when the position at \( L \) is \$. Then the algorithm pops all elements from \( D \) and the initial string \( S \) is fetched.

**Algorithm 1: BWT transformation**

**Input:** String \( S \)

**Output:** BWT(\( S \)) = \( W \)

\[
\text{begin}
\text{int} \ l = \text{length}(S) + 1;
\text{S}.append(\$);
\text{i}=0;
\text{while} \ i<l \text{ do}
\begin{align*}
\text{r}_i &= \text{rotate}(S, \$) \quad \text{// The rotate algorithm permutes the}
\text{characters of the original string and returns the permuted}
\text{string;}
\text{W}.addrow(\text{r}_i) \quad \text{// It adds the permuted row from the previous}
\text{step to the matrix} \ W;
\text{i} &+ 1;
\end{align*}
\text{\textbf{end}}
\text{return} \text{Sorted.}W;
\text{end}
\]

**Algorithm 2: LF Mapping**

**Input:** First (\( F \)), Last column (\( L \)) from BWT

**Output:** \( S \)

\[
\text{D}=0 \quad \text{// Initialize the stack} \ D;
\text{l}=\text{length}(F) \quad \text{// the length of} \ F = \text{length}\text{ of} \ L;
\text{i}=0;
\text{while} \ L[i] = \$ \text{ do}
\begin{align*}
\text{D}.\text{push}(L[i]);
\text{i} &= \text{find.}[L[i]] \quad \text{// find.[] denotes the index number in array}
\text{[]} \quad \text{that the element is. For instance}\text{find.}[S'] = 7;
\text{while} \ D[i] \neq ‘0’ \text{ do}
\begin{align*}
\text{S} &= \text{S} + \text{D}.\text{pop};
\text{\textbf{return}} \text{S};
\end{align*}
\end{align*}
\]

<table>
<thead>
<tr>
<th>( F )</th>
<th>( L )</th>
<th>( F )</th>
<th>( L )</th>
<th>( F )</th>
<th>( L )</th>
<th>( F )</th>
<th>( L )</th>
<th>( F )</th>
<th>( L )</th>
<th>( F )</th>
<th>( L )</th>
<th>( F )</th>
<th>( L )</th>
<th>( F )</th>
<th>( L )</th>
<th>( F )</th>
<th>( L )</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
<td>k</td>
<td>i</td>
<td>i</td>
<td>k</td>
<td>i</td>
<td>k</td>
<td>i</td>
<td>i</td>
<td>k</td>
<td>i</td>
<td>k</td>
<td>i</td>
<td>k</td>
<td>i</td>
<td>k</td>
<td>i</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>a</td>
<td>k</td>
<td>a</td>
<td>k</td>
<td>a</td>
<td>k</td>
<td>a</td>
<td>k</td>
<td>a</td>
<td>k</td>
<td>a</td>
</tr>
<tr>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>s</td>
<td>1</td>
<td>s</td>
<td>1</td>
<td>s</td>
<td>1</td>
<td>s</td>
<td>1</td>
<td>s</td>
<td>1</td>
<td>s</td>
<td>1</td>
<td>s</td>
<td>1</td>
<td>s</td>
</tr>
</tbody>
</table>

Figure 1: The LF mapping process is used to reconstruct the original string \( S \) from the transformed one after applying the BWT operation. Starting with the \$ sign from the \( F \) column, the mapping progressively reconstructs the entire string \( S \). The last column \( L \) is used as a “ladder step” to find the next \( i \)th index in the \( F \) column, which in turn maps to the \( i \)th entry in the \( L \) column. The entire procedure halts when \( L[i]=\$ \).
4.1.3 FM Index. Suffix array vanilla construction has $O(n^2 \log n)$ asymptotic computational cost. This stems from the fact that the $n$ suffixes are first sorted by performing $O(n \log n)$ comparisons and each comparison has cost $n$. Linear time algorithms have been designed by first constructing a suffix tree and then traversing it with a depth first edge in lexicographical order. However, our goal is to storage efficient, meaning we want to eliminate the storage cost of a suffix tree, which practically approximates a constant factor of $20n \in [1, 3, 17, 24]$. We pick up the skew algorithm [20] which is a divide and conquer based algorithm and achieves linear time construction. The approach of the skew algorithm is to recursively divide the suffixes in three groups depending on the position $pos$ of all suffixes: $pos \mod b, h \in [1, 2, 3]$ and then merge the result.

The FM consists of three column arrays. The first one is the $F$ column from the LF mapping, the second one is the $L$ column, which corresponds to the BWT($S$) and the last one corresponds to the suffix array $SA$. $SA$ contains at each row $i$, the position in the original string $S$ of the substring which corresponds to the $i^{th}$ row of the $W$ matrix obtained after applying the BWT transformation. $L = BWT(S)$ can be computed with the formula $BWT(S)[i] = S[SA[i] - 1]$ from the suffix array. Furthermore for the traversal of the LF mapping the unique ranking of each character in each $F$, $L$ needs to be stored in $r_F, r_L$ accordingly. Finally $FM = [F[i], L[i], r_F[i], r_L[i], SA[i]]_{i=1}^{n}$.

The entire challenge is how a user encrypts $FM$ in such a way that an untrusted cloud can correctly reply with the encrypted position on substring queries. We give the cryptographic tools that we use in our protocol in the next subsection. We give an intuition of our approach in the next section, we also highlight some shortcoming thereof and we demonstrate our solution to alleviate it.

4.2 Cryptographic Primitives

4.2.1 Pseudorandom functions (PRF). Let the family of all functions in the universe from a domain $X$ to a range $Y$ be $Func[X, Y]$. A truly random function $f \leftarrow Func[X, Y]$ is chosen randomly from the set of $Func$. The set of these functions is $|Y|^{|X|}$ (gigantic number). It is true that for any random function $f$ with range size $L$ chosen randomly from $Func[X, Y]$, $Pr[f(x) = y] = 2^{-L}$. The randomness is not parametrized neither by the size of $X$ and $Y$ nor by the size of the domain. We define a pseudorandom function $f_k : X \rightarrow Y$ as a function from the set of all functions from $X$ to $Y$ as soon as a particular key $k$ is fixed.

**Definition 4.1.** Let $Func = \{F : X \rightarrow Y\}$ be a function family for all functions $F$ that map elements from the domain $X$ to the range $R$. Then a PRF $= \{f_k : X \rightarrow Y\} \subseteq Func$ for $k \leftarrow K$, where $K$ is the key space.

The security of a PRF is modeled with a game which is known as real or random security game[?]. Intuitively, an adversary $A$ is given access to an oracle that on input $x$ from a domain $X$, flips a coin $b \leftarrow \{0, 1\}$ and if $b = 0$ then it outputs $y = f(x)$, for $f \in Func[X, Y]$, otherwise it outputs $y = f_b(x)$. $A$ issues queries to the oracle polynomially many times on input of the security parameter $\lambda$. Finally $A$ outputs a guess $b'$ for the bit $b$.

The advantage of a probabilistic polynomial time algorithm $A$ in the PRF game is

$$Adv^{PRF}_A = Pr[b \leftarrow \{0, 1\}; b' \leftarrow A(y) : b' = b]$$

**Definition 4.2.** A PRF is computationally secure if all probabilistic polynomially time algorithms $A$ have advantage in the PRF game: $\frac{1}{2} + \epsilon(\lambda)$, for a negligible function $\epsilon$ on the security parameter $\lambda$.

4.2.2 Pseudorandom permutations (PRP). A permutation is a bijective function where the domain and the range are equal. Similarly with the random functions, let $Perm[X]$ to be the set of all permutations for the domain $X$. Then a pseudorandom permutation (PRP) is a randomly chosen permutation from the set $Perm[X]$, keyed under a secret key $k$.

The advantage of a probabilistic polynomial time algorithm $A$ in the PRP game is

$$Adv^{PRP}_A = Pr[b \leftarrow \{0, 1\}; b' \leftarrow A(y) : b' = b]$$

**Definition 4.3.** A PRP is computationally secure if all probabilistic polynomially time algorithms $A$ have advantage in the PRP game: $\frac{1}{2} + \epsilon(\lambda)$, for a negligible function $\epsilon$ on the security parameter $\lambda$.

4.2.3 Symmetric Key Encryption. A symmetric key encryption scheme $SKE = \{Gen, Enc, Dec\}$ consists of three algorithms. $Gen$ takes as input a security parameter $\lambda$ and outputs the secret key $sk$. The probabilistic encryption algorithm $E$ takes as input the secret key $sk$ and a plaintext $x$ form the plaintext space $P$ and outputs the ciphertext $c$. The decryption algorithm $SKE.Dec$ takes as input a ciphertext form the ciphertext space $C$ and the secret decryption key $sk$ and outputs the plaintext $x \in P$. Correctness follows $\iff \forall sk \leftarrow Gen(\lambda^2), SKE.Dec((E(sk, x))) = x, \forall x \in P$. Security is modeled with the standard game based indistinguishability experiment for polynomial probabilistic time adversary $A$.

$Priv_{K,A,SKE}(\lambda)$:

- $A$ has access to the security parameter $1^\lambda$.
- A key $sk \leftarrow Gen(\lambda^2)$ is generated and $A$ can learn encryptions of $x$ of its choice $x \in S \subseteq P$.
- Eventually $A$ outputs $x_0, x_1$ where $|x_0| = |x_1|$, $b \leftarrow$ and $E(x_b, sk)$ is returned to $A$.
- $A$ outputs its guess for $b, b'$.

If $b' = b$ $A$ succeeds and the experiment $Priv_{K,A,SKE}(\lambda) = 1$.

**Definition 4.4.** A symmetric encryption scheme $SKE$ has indistinguishable encryptions if the probabilities $Pr[Priv_{K,A,SKE}(\lambda) = 1] = \frac{1}{2} + \operatorname{neg}(\lambda)$.

5. INTUITION

In this section we provide some intuition about $S^3E$ protocol before delving into its precise description in the follow up section. First we start with showing how the client encrypts the index to allow the cloud process fast encrypted substring queries. Our solution is based on the FM index described in the previous section. The FM index consists of three arrays, which keep track of the $F$, $L$ columns and the encrypted SA suffix array with positions of substrings and not all the suffixes as in [26]. To recap, the user computes the suffix-array $SA$ and the $F$, $L$ columns through the BWT transformation.
### Table 1: Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>σ</td>
<td>Vocabulary size</td>
</tr>
<tr>
<td>λ</td>
<td>Vocabulary</td>
</tr>
<tr>
<td>S</td>
<td>Original string</td>
</tr>
<tr>
<td>f</td>
<td>Substring query</td>
</tr>
<tr>
<td>c</td>
<td>character</td>
</tr>
<tr>
<td>b</td>
<td>Bucket</td>
</tr>
<tr>
<td>n</td>
<td>Size of S</td>
</tr>
<tr>
<td>m</td>
<td>Size of T</td>
</tr>
<tr>
<td>wmn</td>
<td>window size</td>
</tr>
<tr>
<td>sk</td>
<td>salt array</td>
</tr>
<tr>
<td>f</td>
<td>First Column of LF mapping</td>
</tr>
<tr>
<td>t</td>
<td>Last Column of LF mapping</td>
</tr>
<tr>
<td>LLSet</td>
<td>Hash map</td>
</tr>
<tr>
<td>LL</td>
<td>Linked List</td>
</tr>
<tr>
<td>cE, bE</td>
<td>fth character/bucket from F column</td>
</tr>
<tr>
<td>aE, bE</td>
<td>fth character/bucket from L column</td>
</tr>
<tr>
<td>rE1, rE2</td>
<td>Ranking of fth character/bucket in the string S</td>
</tr>
<tr>
<td>rFj</td>
<td>Ranking of fth character from F column</td>
</tr>
<tr>
<td>rLj</td>
<td>Ranking of fth character from L column</td>
</tr>
<tr>
<td>cω</td>
<td>ωth character/bucket from the alphabet, (1 ≤ ω ≤ σ)</td>
</tr>
<tr>
<td>bω</td>
<td>ωth bucket (1 ≤ ω ≤ n−win + 1)</td>
</tr>
<tr>
<td>σ</td>
<td>Security parameter</td>
</tr>
<tr>
<td>F(·)</td>
<td>Pseudorandom function (PRF)</td>
</tr>
<tr>
<td>H(·)</td>
<td>Pseudorandom permutation (PRP)</td>
</tr>
<tr>
<td>SKE = (Gen, Enc, Dec)</td>
<td>Symmetric encryption scheme</td>
</tr>
<tr>
<td>kF</td>
<td>PRF key</td>
</tr>
<tr>
<td>kL</td>
<td>PRF key</td>
</tr>
<tr>
<td>wskL, wskF</td>
<td>TEE keys</td>
</tr>
</tbody>
</table>

### 5.1 First Approach

We give an overview of our first approach. The protocol can be described in two phases: The encrypted index phase and the search phase.

#### 5.1.1 Encrypted Index

To facilitate the reader we split the encrypted index process in two steps: (cf. figure 2) the linked list, which bootsstraps the search procedure on the FM index by the cloud and 2) FM index itself. The notation used for the protocol is given in table 1. For the security of the scheme the user employs lightweight cryptographic primitives: a pseudorandom function $F(·)$, a pseudorandom permutation $H(·)$ respectively and a symmetric encryption scheme $SKE = \{Gen, Enc, Dec\}$. The untrusted cloud, thanks to the LF mapping and the FM index computation does not need to store all the suffixes of a stream $S$ (cf. figure 2).

#### Linked List

The crux of the design is on how to allow fast indexing through a hash table, which means that there should be unique keys derived from the string with repetitive characters. We employ the ranking information $r_c$ of each character along with the character itself. However, when a user is looking for a substring, it does not know the ranking of each character in the substring $T[1...m]$. We mitigate this deficiency by building a linked list $LL_c$ for each character.

The user computes a hash table of linked lists $LLSet$, where each position $LLSet[F_k(c^w)]$, $1 \leq w \leq \sigma$ maps to the linked list $LL_{c^w}$. The number of linked lists equals the number of distinct elements $c$, denoted as $\sigma$ in the data stream $S$, where each symbol $c^w$, $1 \leq w \leq \sigma$ comes from an alphabet $\Sigma$. The hash table is used to fetch all the positions of a character in the stream $S$ from the linked lists $LL_{c^w}$. Each linked list $LL_{c^w}$ stores information concerning the retrieval of the position of $c$ from $S$. More specifically each node in the list stores the following tuple: $(\text{nptr}, \text{addr})$, nptr is a pointer to the next node of the current list and addr is the address of the element $c$ in the FM index.

The first node of each linked list is stored in the $LLSet$ hash table. In order to prevent the adversary who tries to correlate elements from the $LLSet$ with positions in the FM index, we further encrypt each key $F_k(c^w)$ in the $LLSet$ hash map with another key $k_L$ as follows: $F_k(c^w) \oplus F_k(c^w)$. Thus the cloud cannot correlate associations from $LLSet$ to FM attack offline without observing any token. The key of the hash map $LLSet$ at $F_k(c^w) \oplus F_k(c^w)$ maps to the first element of the linked list $LL_{c^w}$, which is encrypted as $(\text{nptr}, \text{addr}) \oplus F_k(c^w)$. As such, the frequency of each character before a search query is hidden.

However, once the cloud receives queries, it can learn the frequency of encrypted characters in the linked list which represent characters of the string. In order to obfuscate frequency analysis on the encrypted index from substring search queries, we pad the data stream with dummy blocks. These dummy blocks make all linked

---

1 Notice that even if we use a one time pad with the same key for two different elements: $a = (\text{nptr}, \text{addr})$, $b = F_k(c^w)$ an adversary by soring the two ciphertexts encrypted under the same key $F_k(c^w)$, learns $ab = (\text{nptr}, \text{addr}) \oplus F_k(c^w)$, which is a one time pad encryption of $(\text{nptr}, \text{addr})$ with key $F_k(c^w)$.\footnote{\textsuperscript{1}}
lists to appear with the same size. The core idea for padding is to produce dummy blocks from the vocabulary \( \Sigma \) depending on the ranking of the most frequent character. E.g: Original stream=abced and \( \Sigma=abcd \), then the dummy blocks equal dc=\( \{a, c, d\} \). Finally
the user chooses uniformly at random \( \text{dpos} \in \{0, \ldots, |dc|\} \) and appends the original string \( S \) at position \( \text{dpos} \) of dc. Following the previous example; if \( \text{dpos}=1 \) then \( S'=acdbdc \). The cloud responds in the final round with the encrypted position \( \text{pos} \). User accepts the result as correct if \( \text{dpos} \leq \text{pos} \leq n-\text{dpos} \) and \( \text{pos} + m < \text{dpos} + n \).

**FM Index Encryption (figure 2).** The second difficulty comes when the cloud tries to traverse the FM index through the LF mapping technique. The encrypted FM index contains unique digests of characters, while the cloud should identify matches from the token \( \text{tk}_{T,S} \), that encodes repetitive characters deterministically. In order to allow the cloud traverse the encrypted FM index, we encrypt the FM as a key value hash table where the key consists of \( F_k(\text{sr}_j) \) and the value is \( F_k(\text{sr}_j)||F_k(\text{r}_j||l_{\text{r}_j}||c_{\text{r}_j})||E(\text{pos}_j) \). Finally the user permutes all the tuples with a secure permutation: \( \Pi_{k_{\text{SA}}} \).

### 5.1.2 Search

**Bootstrap.** First it needs to bootstrap the search by finding the correct candidate positions in the encrypted FM through the linked list, which correspond to all positions in the string \( S \) where the last character of the query possibly exists. From the LLSet hash table it looks for the value with key \( F_k(T[m]) \) at \( F_k(T[m]) \). This value maps to a linked list \( L \), in which each node maps to the encrypted FM tuple \( t_j = t^1_j,t^2_j,t^3_j \) as follows:

\[
\begin{align*}
(F_k(\text{sr}_j) \oplus F_k(\text{r}_j)||\text{sr}_j), \\
F_k(c_{\text{r}_j}) \oplus F_k(\text{r}_j)||\text{sr}_j)||F_k(r_{\text{r}_j}||l_{\text{r}_j}||c_{\text{r}_j}), \\
E(\text{pos}_j))^{\pi_{\text{SA}}}_{j=1}
\end{align*}
\]

In order to decrypt the first element of the linked list the cloud uses \( F_k(T[m]) \) as a key to decrypt \( \langle\text{ptr}, \text{addr} \rangle \) of \( F_k(e^{\text{ptr}}, \text{addr}) \). The cloud uses \( F_k(T[m]) \) and applies an xor operation on the \( F \) column at the ranges that it retrieved from the linked list of the \( c_{\text{m}} \) character \( LL_{C} \) and learns \( F_k(\text{r}_j)||\text{sr}_j) \).

**Iteration.** The cloud uses \( F_k(\text{r}_j)||\text{sr}_j) \) as a key to decrypt the first part of the \( L \) column element \( F_k(c_{\text{r}_j}) \oplus F_k(\text{r}_j)||\text{sr}_j) \) and reveals \( F_k(c_{\text{r}_j}) \). It then fetches the encrypted column as \( k_F(c_{\text{r}_j}), b=F_k(r_{\text{r}_j}||l_{\text{r}_j}||c_{\text{r}_j}) \) in which \( F_k(c_{\text{r}_j})=F_k(T[m-1]) \) and for all nodes from the linked list computes \( k \oplus b \), which is used as a key for the \( F \) column. The procedure terminates when the processed substring character is the first one \( F_k(T[1]) \).

At this point the cloud returns to the user all the encrypted \( E(\text{pos}_j) \) for the substrings. The user decrypts and accepts the result as long as the decrypted position is in the range of original stream without padding.

**Second round.** From the per-character one way function \( F_k \) evaluation of the substring query: \( \text{tk}_{T,S} = C[1],C[2],\ldots,C[m] \leftarrow F_k(T[1\ldots m]) \) and the LF mapping the protocol leaks to the cloud in ciphertext the exact differences of the positions of two encrypted substrings as long there are unique matches. More specifically, the number of iterations in the LF mapping traversal (algorithm 2), reveals how many positions two substrings they differ, as long as there is unique match in \( S \). Eventually, an untrusted cloud can decrypt the entire encrypted \( SA \) array, which contains encrypted positions of all substrings in \( S \), since it knows its addresses. To circumvent the leakage we first use two different permutations to permute the tuples \( t_j: \Pi_{k_{\text{SA}}} \) as long as there is unique match in \( S \).

5.2 Improved scheme

5.2.1 Shortcoming. Recall that during the encryption of the index, the user adds dummy blocks at the linked lists in order to alleviate frequency attacks. Namely after issuing a query token \( \text{tk}_{T,S} \) the client reveals to the cloud the key \( F_k(T[m]) \) in order to locate the head of the list which corresponds to the key \( F_k(T[m]) \) at \( F_k(T[m]) \). The cardinality of the list corresponds to the frequency of that character in the original string \( S \). In our first approach we address this problem by adding dummy blocks in all linked lists in order their size to equal the size of the longest one.

The aforementioned technique protects the client from frequency attacks on the original string and imposes low overhead in case of a string drawn from a distribution with homogeneous frequency characters. However in a more skewed dataset with characters having broad frequencies, then the technique of dummy characters can drastically affect the efficiency of the system. More specifically the dummy characters may double the size of the final size of the index, thus degrading the storage overhead and the computational efficiency of the client.

5.2.2 Bucketization of characters. **Index.** The entire procedure is similar to the first approach but instead of operating on single characters everything operates on buckets. Now the \( F \) and \( L \) column arrays correspond to buckets and the input to the \( F_k \) is not a single character \( c \) but a bucket \( b \). We use the same notation conventions with the first approach but instead of operating on characters we operate on buckets, i.e: we denote by \( b^1 \) the first bucket of string,
We give the full details of our substring searchable symmetric encryption protocol, which alleviates the storage overheads of previous approaches. Our protocol has the following properties:

- KeyGen(1^λ): This algorithm runs by the user takes as input the security parameter 1^λ and generates random keys \( k = (k_1, k_1, k_π, k_{π+1, 1}, k_{π+2}) \) for a PRF \( F_k : \{0, 1\}^λ \times \{0, 1\} \rightarrow \{0, 1\}^λ \), a PRF \( π_k : \{0, 1\}^λ \times \{0, 1\} \rightarrow \{0, 1\}^λ \) and a symmetric encryption algorithm \( \text{SKE} = (\text{Gen}, \text{Enc}, \text{Dec}) \). Finally, it outputs \( k \) to the user. For the generation of the key, we assume a source of randomness \( \mathcal{R} \) and a pseudorandom generator \( G \) seeded with

\[ s_π \leftarrow \mathcal{R}, s_{π+1} \leftarrow \mathcal{R}, s_{π+2} \leftarrow \mathcal{R}, s_{π+3} \leftarrow \mathcal{R}; \]

\( k_1, k_1, k_π, k_{π+1, 1}, k_{π+2} \) such that \( G(s_1), G(s_1), G(s_π), G(s_{π+1}), G(s_{π+2}) \)

- \( \text{SKE} \leftarrow \text{PreProcess}(k, S) \): User owns a stream \( S \), which contains characters \( c \in \Sigma \). \( S \) has \( n \) characters. Let \( m_{\text{max}} \) be the cardinality of most frequent bucket and \( b_z \), the frequency of bucket \( b_z \).

(1) \( \text{Parses} \) the string \( S \) as buckets: \( b_{z} \) for all \( z \geq n \text{win} + 1 \), each of size \( \text{win} \) characters and \( z \) in total distinct buckets:

\[ \text{for } z = 1; z + \text{win}; z = n - \text{win} \]

\[ b_z = (z + \text{win} \leq |S|)? S[z..z + \text{win}] : S[z..n - z] \]

(2) \( \text{Chooses} \) dummy buckets \( d_c = \sum_{j=1}^{k} m_{\text{max}} - b_z \) that constitute a dummy stream. The user chooses uniformly at random \( d_{\text{pos}} \leftarrow \{|d_c|\} \) and appends the original string \( S \) at position \( d_{\text{pos}} \) of \( d_c \).

(3) \( \text{Computes} \) the suffix array \( \text{SA} \) and the \( F, L \) columns on input the buckets \( B \) and stores them as the FM index: \( \text{FM} = F||L, \text{SA} \).

(4) \( \text{Encrypts} \) all bucket \( B = \text{SKE} \cdot E(k_c, b_z), 1 \leq z \leq n + k - \text{win} + 1 \)

(5) \( \text{Permutates} \) \( B'[j] = B[k_{\text{pos}}(j)] \)

(6) \( \text{Encrypts} \) elements of \( \text{SA} \) with \( \text{SKE} \cdot E(k_c, \text{SA}(i)), 1 \leq i \leq k - \text{win} + 1 \)

(7) \( \text{Applies} \) the PRF to each element of \( F \) as follows:

\[ F[i] = F_k(b_z) \oplus F_k(r_z)||b_z \]

(8) \( \text{Computes} \)

\[ L[i] = F_k(b_z) \oplus F_k(r_z)||b_z \]

(9) \( \text{Applies} \) a pseudorandom permutation \( \pi_{\text{pos}} \) to the tuples:

\[ t_0 = F_k(b_z) \oplus F_k(r_z)||b_z \]

\[ t_1 = F_k(b_z) \oplus F_k(r_z)||b_z \]

using \( k_{\text{pos}} \), and \( \pi_{\text{pos}} \) user permutes \( E(k_c, \text{SA}(i)) \), for \( i = 1, \ldots, n + k - \text{win} + 1 \):

\[ \pi_{\text{pos}}(t_0) = \pi_{\text{pos}}(t_1) = \pi_{\text{pos}} = \text{FM} \]

(10) \( \text{For every distinct bucket in } F[i] = F_k(b_z) \oplus F_k(r_z)||b_z \)

the user \( \text{initiates} \) a linked list \( \text{LL}_c \) and at each node stores \( \text{LL}_{c, \text{pos}} \) for the next node of the list and \( \text{LL}_{c, \text{addr}} \) which points to the tuple \( t_i \) with a matching \( F_k(b_z) \). Finally it \( \text{encrypts} \) the first element of each linked list \( \text{LL}_c \) with \( F_k(b_z) \) : \( \text{nptr, addr} \rightarrow F_k(b_z) \).

(11) \( \text{Stores} \) the head pointers of the collections of all linked lists in a hash table \( \text{LS}\text{Set} \) with key \( k = F_k(b_z) \oplus F_k(b_z) \) and value \( v \) a pointer to the head of the list \( \text{LL}_c \), which stores information about the \( F_k(b_z) \) character, meaning all its positions to the encrypted FM index.

(12) \( \text{Finally outputs} \) \( \text{SES} = (\text{LS}\text{Set}, \text{LL}_c, \text{FM'}, B') \) and keeps only the keys \( k = (k_1, k_1, k_{π+1, 1}, k_{π+2}) \).

\( \text{tk}, \text{S} \leftarrow \text{SrchToken}(k, T[1..m]) \): This algorithm takes as input the secret substring search key \( k \), a string \( T[1..m] \) and outputs a trapdoor to search for the string \( T \) on data stream \( S \), through \( \text{SES} \):

(1) \( \text{Parse} \) \( T[1..m] \) to buckets \( T_b = (T_{b_i})_{i=1}^{m-\text{win}+1} \) of size \( \text{win} \):

\[ \text{for } i = 1; i + \text{win}; i = m - \text{win} \]

\[ T_{b_i} = (i + \text{win} \leq |S|)? T[i..i + \text{win}] : T[i..m - i] \]

(2) User with his secret PRF key \( k_f \) computes

\[ \text{tk}, \text{S} = C[1], C[2], \ldots, C[m - \text{win}] \]

and forwards \( \text{tk}, \text{S} \) to the cloud.

(3) \( \left( i, \ldots \right) \leftarrow \text{Search}(\text{tk}, \text{S}, \text{SES}) \): The cloud \( \text{parses} \) the token query \( \text{tk}, \text{S} = C[1], C[2], \ldots, C[m - \text{win}] \) and searches the position in \( S \) from the encrypted index \( \text{SES} \) as follows:

(1) if \( |\text{tk}, \text{S}| \) mod \( \text{win} \) == 0

\[ x = m - \text{win}, \text{flag} = 0 \]

else
We illustrate the security of the scheme pertaining to definition 2.2. User splits the original data stream \( S \) in consecutive buckets of size \( \text{win} \) by moving each time the window to the right one character. More specifically, we show the existence of a simulator \( S \), who has access to the leakage function \( L \) and produces indistinguishable views to an adversary \( A \). Conceptually the proof demonstrates that an adversary \( A \), who can be a semi-honest cloud cannot learn more information from what it can be leaked in an ideal work without malicious behaviors. During the encryption of the index users write to the \( LLSet \) linked list and to a three dimensional array \( FM \). \( FM \) consists of three columns: \( F \), \( L \) from the \( BWT \) transformation, which are used to traverse the string, and the \( SA \) column, which consists of encrypted positions of the corresponding suffixes. To avoid leakage of frequencies users pad each linked list \( LLSet \). In order to impede off-line traversal of the index, the user further encrypts the header of each linked list. Moreover, two different permutations \( \Pi_{k_{win}} \) are applied first on all the rows of the first two columns \( F, L \) and then to the \( SA \) column to prevent an adversary from correlating encrypted positions to intermediate results after having obtained a token. I.e. without the permutation, \( A \) after obtaining a token can stop the search to a substring of the queried substring and learn the associated encrypted position from \( SA \) column. After \( A \) observing \( LLSet \) and \( FM \) it learns the size of the padded string \( n' \).

During the search phase the adversary \( A \) observes encrypted tokens. Between two encrypted tokens \( A \) can identify similarity in between two tokens: i.e: whether two identical token have been issued, or identical encrypted buckets in between the tokens. Then we point out two different cases:

1. **Size of token multiple of win**: The user leaks to the cloud how many matches exist in the string \( S \) for the queried token, the encrypted cells at \( SA \), which correspond to the matched positions.
2. **Size of token not a multiple of win**: The user in case of a token, whose size is not a multiple of the size win of each bucket, needs to verify the correctness of the potential encrypted match(s). As such the cloud returns the permuted possible position pos. User decrypts and applies \( pos' = \Pi_{k_{win}}(pos) \) and asks the cloud to return the bucket at position \( pos' \). The cloud learns the encrypted position of the last bucket of the token and its encrypted text.

Integrity guarantees of the data and the index are assured thanks to the use of authenticated symmetric cryptography. Before stating our theorem concerning the security of \( S \) we formally define our leakage function \( L \) from our aforementioned analysis.

**Definition 7.1.** A leakage function \( L \) for a \( S^2E \) scheme comprises the following three leakage functions:

- **PreProcess Leakage**: \( L_1 \) includes the padded size of the data stream \( n' = |S| \).
- **SrchToken Leakage**: The SrchToken Leakage \( L_2 \) reveals the length of the token \( |tk| \), how many common characters reside in it and similarity patterns between different tokens.
- **Search Leakage**: \( L_3 \) leaks how many times a substring token \( tk \) exists in the padded string \( S \) with dummy blocks.
- **Intermediate**: Leakage \( L_4 \) leaks intermediate addresses of the \( F \) and \( L \) columns of a query.
- **Access pattern**: Leakage \( L_5 \) leaks the addresses of the \( SA \) cells for a fixed query.
To eliminate $L_2, \ldots, L_5$ leakage profiles an expensive ORAM scheme can be used, however it is out of the scope of the paper, as it will increase communication and storage overhead drastically.

**Theorem 7.2.** Let $F_k, \Pi_{k^*}, SKE = \{Gen, Enc, Dec\}$ be a pseudorandom function, a pseudorandom permutation, a semantically secure symmetric encryption scheme respectively, then our substring searchable symmetric encryption scheme $SSE$ is adaptively $L$-semantically secure.

**Proof.**

In the $Real_{\Pi_{k^*}}^{SSE}(\lambda)$ world (cf. 2.2) the adversary can obtain the encrypted index and encrypted keywords of its choice. In the beginning the Challenger selects the size of the buckets win and uniformly at random keys $k = (k_1, k_2, k_3, k_4, c)$ for a PRF $F_k : \{0, 1\}^c \rightarrow \{0, 1\}^\mu$, a PRP $\Pi_{k^*} : \{0, 1\}^\lambda \rightarrow \{0, 1\}^\nu$ and a symmetric encryption algorithm $SKE = \{Gen, Enc, Dec\}$.

Upon receipt of a stream $S$ of size $n$, the Challenger employs the $SKE \leftarrow PreProcess(k, S)$ as presented in section 6 and forwards $SKE$ to $A$. We distinguish between matching $q^m$ and non-matching queries $q^{nm} : q = \bigcup q^{nm} q^m$. We assume for the ease of readability that adversary issues only matching queries $q^m$. During the Search phase, $C$ has also access to $L_2$=similarities between tokens, $L_3$=# times a token exist in the substring. $C$ upon receipt of $q$ checks in a table $QT$ whether $q \in QT$. If so $C$ fetches the corresponding token $tk_S$ and forwards it to $A$. If this is the first time for $q$ then $C$ computes $tk_{S,S} = C[1], C[2], \ldots, C[m]$. Finally $A$ receives $t = (tk_1, tk_2, tk_3, \ldots, tk_9)$ for each substring query.

Within a sequence of hybrid games we show the indistinguishable transformation of $Real_{\Pi_{k^*}}^{SSE}(\lambda)$ game to eventually the $Ideal_{\Pi_{k^*}}^{SSE}(\lambda, S)$ game, which concludes the proof. The simulator $S$ computes the simulated encrypted index $SSE^* = (LLSet^*, LL^{k*}, FM^*)$ as follows:

- **Game0:** This game is equivalent with the $Real_{\Pi_{k^*}}^{SSE}(\lambda)$ game.
- **Game1:** This game behaves as the $Real_{\Pi_{k^*}}^{SSE}(\lambda)$ game with the difference that $S$ does not have access to $S$. The simulator through the $L_1$ leakage function builds the substring encrypted structure $SES$ as follows: We assume the existence of an algorithm $S \leftarrow Build(n', str)$, which takes as input $n' \in \mathbb{N}$ and the structure $str = [b]_{i=1}^{n'}$, $b \in \Sigma^*$ and outputs a bitstring of length $n'$, from a vocabulary $\Sigma^*$. Notice that as in the real game the valid length of the original stream is not revealed and only the length of the string after the padding $n'$ is leaked. $S$ selects uniformly at random keys $k = (k_1, k_2, k_3, k_4, c)$ for a PRF $F_k : \{0, 1\}^c \rightarrow \{0, 1\}^\mu$, a PRP $\Pi_{k^*} : \{0, 1\}^\lambda \rightarrow \{0, 1\}^\nu$ and a symmetric encryption algorithm $SKE = \{Gen, Enc, Dec\}$ and runs $SKE \leftarrow PreProcess(k, Build(L_1))$. $S$ uses $F_k$ to evaluate bit strings of length $c_n$: $L_2(q) = c_n$. For the simulation of the tokens and its responses, $S$ uses the leakage obtained from $L_2 \ldots L_5$.

- **Game3:** This game behaves similarly with Game2, but we replace the $F_k$ with a real random function which is evaluated through access to an oracle $O^{RE}(\mu, v)$. We write $Game_i \approx Game_j$ to denote that the view of probabilistic polynomial time adversary $A$ is indistinguishable between the output of $Game_i$ and $Game_j$.

**8 PERFORMANCE**

In this section we present our implementation results. We demonstrate the practicability of $SSE$ with benchmark experiments, comparing our results with the scheme of Chase et al. [7], in order to validate the claims of our performance improvements. To accomplish the comparison we also implemented the suffix tree based construction of [7] called hereafter $ST$.

### 8.1 Implementation

Our comparison is based on two metrics: a) **storage overhead** for the encryption of the index and the computation of a substring query as an encrypted token, b) **computation time** of each operation. The reported computation times for each experiment are taken as the average of 100 trials. As we implemented both schemes on the same machine, which simulates both the client and the server, we can derive accurate and fair observations about the performance of the protocols on real metrics. In the real world the server can be implemented in a more powerful machine, however this does not change the storage overhead or the computation performance fraction of both schemes.

Thanks to our encrypted suffix array construction, we achieve a storage improvement by a factor of 1.71 for the DNA sequence data stream and 1.57 for the enron email data. This occurs first...
because of the extra information a node of the suffix tree should keep (leaf nodes, parent nodes, auxiliary information for the substring of the path in the tree) and due to the dummy nodes policy, which increases the size of the tree. Subsequently that affects the computation time for the computation of the encrypted index with a $\approx 4x$ blowup on average for data sets of size $10^6$.

8.2 Benchmarks

8.2.1 Index. In tables 3, 4, we depict the storage and computational overhead incurred by the computation of the encrypted index using the suffix array in $S^3E$ using different window sizes for the buckets and the suffix tree in ST scheme of [7].

We observed an increased overhead in the size of the encrypted index for the ST scheme [7], compared with ours as expected. On average, over all the the data sizes, for different data sets the gain of $S^3E$ over ST approximates a factor of 1.64. However, for realistic big data streams consisting of $10^6$ the gain reaches a factor of 2. Even though the computation of the encrypted index happens only once, the storage overhead incurred by its encryption is of more crucial importance than its computation time. A limited storage device is not capable of computing the encrypted index if that comes at an increased communication overhead.

We also measured the computation time of the encrypted index in both schemes in tables 3, 4. The $S^3E$ index construction time outperforms ST. Apart from the extra dummy blocks and the increased size of the suffix tree compared with that of a suffix array, the increased computation cost stems from the way ST encrypts the suffix tree.

8.2.2 Query Encryption. We run experiments in order to compute the storage overhead during the SrchToken phase. The token consists of a sequence of buckets from a vocabulary: be it characters from emails or characters from DNA sequence. As the query encryption is not affected by the distribution of the underlying characters and for compactness we choose to present results only from the DNA sequence. We also observed tiny differences at the

Because of the extra information a node of the suffix tree should keep (leaf nodes, parent nodes, auxiliary information for the substring of the path in the tree) and due to the dummy nodes policy, which increases the size of the tree. Subsequently that affects the computation time for the computation of the encrypted index with a $\approx 4x$ blowup on average for data sets of size $10^6$.

8.2 Benchmarks

8.2.1 Index. In tables 3, 4, we depict the storage and computational overhead incurred by the computation of the encrypted index using the suffix array in $S^3E$ using different window sizes for the buckets and the suffix tree in ST scheme of [7].

We observed an increased overhead in the size of the encrypted index for the ST scheme [7], compared with ours as expected. On average, over all the the data sizes, for different data sets the gain of $S^3E$ over ST approximates a factor of 1.64. However, for realistic big data streams consisting of $10^6$ the gain reaches a factor of 2. Even though the computation of the encrypted index happens only once, the storage overhead incurred by its encryption is of more crucial importance than its computation time. A limited storage device is not capable of computing the encrypted index if that comes at an increased communication overhead.

We also measured the computation time of the encrypted index in both schemes in tables 3, 4. The $S^3E$ index construction time outperforms ST. Apart from the extra dummy blocks and the increased size of the suffix tree compared with that of a suffix array, the increased computation cost stems from the way ST encrypts the suffix tree.

8.2.2 Query Encryption. We run experiments in order to compute the storage overhead during the SrchToken phase. The token consists of a sequence of buckets from a vocabulary: be it characters from emails or characters from DNA sequence. As the query encryption is not affected by the distribution of the underlying characters and for compactness we choose to present results only from the DNA sequence. We also observed tiny differences at the query encryption time and the size of the bucket. We observe a reasonable increase in both the size of the encrypted query (figures 3, 4) and its computation time as the query size increases. $S^3E$ outmatches in query computation time due to the way the query is encrypted in ST scheme: Namely for each character $2$ PRF, and one block cipher is invoked, while in $S^3E$ only one PRF is invoked. The storage overhead of ST also outgrows faster since the substring is encrypted recursively and not by character. That is, the token: $T[1], T[2], ..., T[m]$ is encrypted as $ct_1 = PRF(T[1]), k_1 = PRF_2(T[1]), ct_2 = PRF(T[1], T[2]), k_2 = PRF_2(T[1]|T[2])$ and so on. Finally the client forwards to the cloud: $(Enc_k(ct_i))_{i=1}^m$.

8.2.3 Response Overhead. In figures 5, 6 we discern a slight outperformance of $S^3E$ compared with ST in terms of substring response time. For the experiments we computed tokens of various lengths and perform a search on DNA streams of different sizes.
We perform a comparison of our scheme with existing solutions (cf. Table 5). We analyzed the search running time in asymptotic complexity, index space requirements both in the plaintext and in the ciphertext space, query size, variable length capability, rounds of communication and search leakage. Since our scheme competes mostly with [7] here we further elaborate its cost analysis from table 5.

**Search.** Thanks to the usage of encrypted dictionary the cost of searching a $m$ length string is $O(m+k)$, where $k$ denotes the number of occurrences. However, due to the extra dummy blocks the search cost is increased to $O(m + k + \sum_{i=1}^{n} (\max_b - f_b))$, where $\max_b$ is the most frequent bucket and $f_b$ the frequency of bucket $b$.

**Index.** For the index space complexity, we analyzed the space requirement in the plaintext space and in the ciphertext space. For the plaintext space analysis we assume, that a pointer or integer requires 4 bytes. Recall that a suffix tree has $n$ leaves, at most $n-1$ internal nodes and at most $2n-2$ edges. Thus, for a naive suffix tree implementation we need 2 pointers for each leaf: one for the parent node and one for its position to the original stream, resulting in $8n$ bytes. Four pointers for each internal node: one for the parent node, one for each leftmost child, one for the right sibling and one pointer for the suffix link, which reduces the search time during a substring query. The total storage cost for the internal nodes is $4 \times 4n = 16n$. For each edge, suffix trees allocate one pointer for the frequency of bucket $b$.

The client computes and encrypts the index and uploads it to the cloud. The cloud simulated in the same machine runs the search algorithm, and we computed the total search time. We perceived in both schemes, that for considerable smaller than $10^6$ elements the running time tends to be independent on the size of the substring token. For a one million data stream there is a notable increased response time compared with the smaller data sets and there is an increased size of the encrypted index in [7] with dummy blocks, based on a suffix tree data structure.

### Table 3: [7] Index storage and computational overhead for variable size data sets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$10^2$ Storage</th>
<th>$10^9$ Time</th>
<th>$10^6$ Storage</th>
<th>$10^9$ Time</th>
<th>$10^7$ Storage</th>
<th>$10^9$ Time</th>
<th>$10^8$ Storage</th>
<th>$10^9$ Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA</td>
<td>60KB</td>
<td>0.44s</td>
<td>567KB</td>
<td>1.07s</td>
<td>6.4MB</td>
<td>9.66s</td>
<td>63MB</td>
<td>89.50s</td>
</tr>
<tr>
<td>Enron</td>
<td>57KB</td>
<td>0.42s</td>
<td>562KB</td>
<td>0.97s</td>
<td>6.5MB</td>
<td>9.07s</td>
<td>62MB</td>
<td>85.74s</td>
</tr>
</tbody>
</table>

### Table 4: $S^3E$ Index storage and computational overhead for variable size data sets and buckets window size.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$10^2$ Storage</th>
<th>$10^9$ Time</th>
<th>$10^6$ Storage</th>
<th>$10^9$ Time</th>
<th>$10^7$ Storage</th>
<th>$10^9$ Time</th>
<th>$10^8$ Storage</th>
<th>$10^9$ Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>win = 2</td>
<td>37KB</td>
<td>0.26s</td>
<td>410KB</td>
<td>49s</td>
<td>4.2MB</td>
<td>3.30s</td>
<td>42MB</td>
<td>29.96s</td>
</tr>
<tr>
<td>win = 25</td>
<td>38KB</td>
<td>0.17s</td>
<td>401KB</td>
<td>48s</td>
<td>4.1MB</td>
<td>3.46s</td>
<td>40MB</td>
<td>29.78s</td>
</tr>
<tr>
<td>win = 25</td>
<td>29KB</td>
<td>0.11s</td>
<td>371KB</td>
<td>43s</td>
<td>3.7MB</td>
<td>2.34s</td>
<td>37MB</td>
<td>26.12s</td>
</tr>
</tbody>
</table>

**Figure 6: Response time for ST [7] scheme**

The client computes and encrypts the index and uploads it to the cloud. The cloud simulated in the same machine runs the search algorithm, and we computed the total search time. We perceived in both schemes, that for considerable smaller than $10^6$ elements the running time tends to be independent on the size of the substring token. For a one million data stream there is a notable increased response time compared with the smaller data sets and there is a proportional increment in time with respect to the size of the token. In exact times, $S^3E$ surmounts ST [7] for the computation of the response at the cloud side. This outperformance is due to the increased size of the encrypted index in [7] with dummy blocks, based on a suffix tree data structure.

### Table 5: Performance comparison of encryption schemes.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$10^2$ Storage</th>
<th>$10^9$ Time</th>
<th>$10^6$ Storage</th>
<th>$10^9$ Time</th>
<th>$10^7$ Storage</th>
<th>$10^9$ Time</th>
<th>$10^8$ Storage</th>
<th>$10^9$ Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA</td>
<td>60KB</td>
<td>0.44s</td>
<td>567KB</td>
<td>1.07s</td>
<td>6.4MB</td>
<td>9.66s</td>
<td>63MB</td>
<td>89.50s</td>
</tr>
<tr>
<td>Enron</td>
<td>57KB</td>
<td>0.42s</td>
<td>562KB</td>
<td>0.97s</td>
<td>6.5MB</td>
<td>9.07s</td>
<td>62MB</td>
<td>85.74s</td>
</tr>
</tbody>
</table>

For the storage space computation during the encryption of the index, be it suffix tree or suffix array, we exclude a per byte comparison and we assume a ciphertext comparison. The encryption of the index is based on the translation of the suffix tree to an encrypted dictionary. Thus, all the extra pointers of the suffix tree
are excluded. Following the protocol from [7], the user encrypts 2n substrings which are equal to the number of edges of the suffix tree plus n leaves and n characters of the original stream, resulting in 4n encryptions. In our solution thanks to the FM mapping the user sends the encrypted suffix array, plus two more n size arrays for the FM index construction; one for the F column of the index and one for the L column. In the end it uploads 4(n + Σ_{j=1}^{m} max_h - f_{h}) encrypted values to the total.

**Query.** For the query size, we assume a block cipher of size θ and a substring query of size m. In [7] the substring is encrypted incrementally: for the substring “abc” user encrypts separately E(a), E(ab), E(abc). As such, for big substring queries as in DNA queries, the number of ciphertexts exceeds the number of the substring m. The total number of encryptions equals \( \frac{1}{4} + \frac{2}{3} + \cdots + \frac{m}{m+1} \) during the first round. At the last round the user asks for the positions of each character separately augmented by a factor of m the substring size. In S3E the substring query has only per character encryptions of each character in the first round plus a ciphertext for the last round. Our solutions also allows variable size substring queries, since the size of the substring query is decoupled from the scheme and can be defined online during the query phase as in [7].

**Rounds.** Regarding the rounds of communication, S3E can return the substring search results in 2 rounds of communication or in 3 when the query size is not a multiple of the bucket size win. During the first round, the client sends an encrypted substring query and the cloud responds with the encrypted addresses of the corresponding suffix array positions. At the second round the client decrypts the permuted position of the suffix tree and asks the cloud for the unpermuted encrypted position. At a third round the protocol is performed when a query is not a multiple of the size of the bucket, whereby the client verifies the correctness of the possible match.

**Security Leakage.** Concerning the search leakage, Chase et al. [7] scheme leaks the search pattern, meaning an untrusted could identify similarities between two or more substring search queries. Moreover, the scheme reveals the query prefix pattern, which leaks whether a node has been visited for a previous substring in the suffix tree, the index intersection pattern which allows the cloud to learn if the returned index position has already been asked and finally, the leaf index leaks when any of the returned positions of the tree leaves have been previously queried. Since in S3E we avoid the use of a suffix tree, S3E does not leak the leaf pattern. We inherit though, the index intersection pattern, which reveals if any returned index has been returned in a previous query. As in [7] our scheme reveals also the cloud differences of the indexes when a user asks for substrings that do they differ in one position and there is only a single position in the original stream S. Both schemes employ a padding policy to add dummy blocks in order to obfuscate the structure of the index and the stream. S3E is also adaptively secure under the real-ideal simulation paradigm. We also use an authenticated encryption scheme in order to assure the integrity of the messages.

### Table 5: Comparison of existing substring searchable encryption protocols.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCI[7]</td>
<td>( O(m + k) )</td>
<td>( 2n )</td>
<td>( 4(n + \sum_{j=1}^{m} \sigma - child(i)) )</td>
<td>( m )</td>
<td>( m )</td>
<td>( 3 )</td>
</tr>
<tr>
<td>PF[NIS][11]</td>
<td>( O(n) )</td>
<td>( \infty )</td>
<td>( m )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>FFN[12]</td>
<td>( O(m + m \cdot \sigma) )</td>
<td>( \infty )</td>
<td>( m )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>S3E</td>
<td>( O(m + k) )</td>
<td>( 4n )</td>
<td>( 4(n + \sum_{j=1}^{m} \max_h - f_{h}) )</td>
<td>( m )</td>
<td>( 1 )</td>
<td>( 2, 3 )</td>
</tr>
</tbody>
</table>

Table 5: Comparison of existing substring searchable encryption protocols. Index space is further categorized in plaintext space index storage (PS) and ciphertext space (CS). The overhead of [11] and [12] is undefined as the schemes do not take advantage of any auxiliary data structure for efficient substring search. For the query complexity we analyzed its size in terms of two separated phases: at the first round (FR) of the protocol and the last one (LR), in case of multiple rounds protocols. VLS denotes variable length substring search and SP the search pattern leakage: QPP: Query prefix pattern, IIP: Index intersection pattern, LIP: Leaf intersection pattern.

### 10 Conclusion

We designed and analyzed a substring searchable symmetric encryption protocol S3E, which achieves better storage performance than state of the art work [7]. The idea of our protocol is to leverage the self-indexing mechanism of FM index, which stores only n integer positions of its substrings. Our protocol is provably secure under the real-ideal paradigm. We also implemented our protocol and compared it with the state of the art work [7], showing its notable performance improvement in terms of storage overhead and computation time.

As part of future we will investigate solutions for substring queries on encrypted data, which further reduce the search time by tolerating accuracy. This is achieved with approximation algorithms. To the best of our knowledge, at the time of this writing, the literature has not addressed how to perform approximation queries for substring queries on encrypted data. Moreover, we seek to look for solutions, which scale better in terms of computational time by leveraging parallelization: Due to the large amount of data produced in the stream of a DNA sequence, the client may outsource the task of computing the FM index in different servers, each one holding a portion of the data stream. It is becoming challenging how those servers can compute the encrypted index. Recent progress in the area of algorithms has shed some light [36], but tweaking the algorithms to operate on encrypted data without leaking more than it is accepted, needs further research.

### References


