Storage Efficient Substring Searchable Symmetric Encryption

Iraklis Leontiadis, Ming Li

New Jersey Institute of Technology and University of Arizona, USA
{leontiad@njit.edu,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 suffix array which allows optimal storage cost $O(n)$ with small hidden factor at the size of the string n. On top of that we build an encrypted index that allows the server to answer substring queries without learning neither the query nor the result. We identify the leakages of the scheme following the work of Curtmola *et al.* [12] and we analyze the security of the protocol in the real ideal framework. Moreover, we implemented our scheme and the state of the art protocol [9] to demonstrate the performance advantage of our solution with precise benchmark results. We improved the storage overhead of the encrypted index by a factor of **1.8** and the computation time thereof **4** times on 10^6 character data streams.

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. Uploading encrypted data however, renders operation on it infeasible. Downloading, decrypting and running the operation on plaintext data, cancels the advantages, that the cloud offers for large storage and computational efficiency. Usually users need to perform a search on their data. Tailored protocols for secure searchable encryption have been proposed 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 containing a single, multiple or a boolean function of keywords, without compromising index, query, and documents privacy. Apart from their theoretical consideration in the literature, quite a few companies adopt this model to offer searchable encryption schemes over encrypted data [3, 10, 11, 26, 33, 39, 43].

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.

The variety of applications for *substring based queries* spans in health-care analysis [14] and surveillance of malicious behavior. In surveillance based applications an authority through espionage, logs chat rooms of suspicious conversations for criminal activities. The goal of the authority is to identify the involved chat room participants. Suspicious parties at some point post identifiable information such as secret url pages. Authority keeps track of the pages and coerces the cloud service through jurisdictional power to reveal if this specific url (substring) obtained by a chat room is part of their logs in order to identify the suspicious participants. This is accomplished by compelling malicious users to reveal their secret key, which in turn has been used to encrypt and upload data streams of logs in order to comply with the policy. 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 protected. Nowadays, the

sequencing process is possible thanks to the progress of computers. Online services offer DNA sequencing to institutions and individuals. 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 would consist of all the possible substrings of a stream of data of size n and the encrypted data would be the positions of the substring. Recently Chase *et al.* [9] 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, 5, 23, 31] 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 [17] 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.
Idea. 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 succeed decreasing the storage need for the construction of the index. After encrypting the suffix tree of Chase *et al.* scheme [9], the encrypted structure leaks a lot 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 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. The second factor which lends us to less storage and subsequently communication efficiency is the dummy blocks policy which is used to hide the structure of the suffix tree in [9]. Our dummy node policy to obscure the encrypted suffix array rests only on the frequency of the most the most frequent character. Namely, we fill up the original string with characters such that the frequency of each character is the same.

In this paper we design and analyze a storage efficient Substring Searchable Symmetric Encryption ($\mathsf{S^3E}$) protocol with minimal leakage and variable size of substrings. We follow a different approach of 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 without the need to store for the original stream of data. Its form resembles the suffix arrays with some additional extra steps, thus we are taking its computational cost for "free", after building the suffix array. The main contributions in the paper are summarized as follows:

Contributions:

- *Storage efficient Substring Searchable Symmetric Encryption* ($\mathsf{S^3E}$): 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 substring searchable 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, which leaks less than the scheme in [9], thanks to the suffix array construction.
- *Prototype implementation*: We implemented our protocol and the state of the art work in [9]. 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

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 the core idea of our solution. The full protocol description is presented in section 6. We then investigate the security and the costs of the proposed scheme for storage efficient substring searchable symmetric encryption. In section 7 we present our prototype implementation results. Finally, we conclude in section 8.

2 Problem Statement

In this section we formalize first 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 an 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 Σ of size $\sigma = |\Sigma|$. We say that a substring T occurs in S if there exists $s : 1 \leq s \leq n - m$ and $S[s + 1...s + m] = T[1...m]$, meaning that $S[s + j] = T[j], 1 \leq j \leq m$ (cf figure 1).

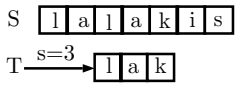

Fig. 1: Pattern matching

Naive algorithms for pattern matching achieve $O(n)$ on search time and 0 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 [28] achieves $O(n-m+1)$ search time and $\Theta(m)$ preprocessing amortized cost. In a similar trajectory Knuth-Moris-Pratt [29] has $\Theta(n)$ search complexity and $\Theta(m)$ preprocessing time. Boyer-Moore pattern matching technique [4] 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 substring are computed along with their positions in the string S, be it suffix tree[32, 42, 44] or suffix array [31]. Suffix tree though has a more expensive space efficiency due to the extra information the suffix tree has to keep [1, 5, 23, 31]. 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 approach which has a much simpler storage cost which approximates $4n$ [1].

We redraw upon the *queryable encryption* syntactical definition of [9], 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 $\mathcal{F}$ 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 introduction. 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 distinction between the index, and the data structure that holds the data (files with keywords). For these reasons we rewrite the functional definitional framework for substring searchable encryption.

Definition 1. *A Substring Searchable Symmetric Encryption scheme (*$\mathsf{S^3E}$*) is a collection of four polynomial time algorithms (*$\mathsf{KeyGen}, \mathsf{PreProcess}, \mathsf{SrchToken}, \mathsf{Search}$*) defined as follows:*

- $\mathsf{k} \leftarrow \mathsf{KeyGen}(1^\lambda)$*: 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.
- $\mathtt{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S)$*: This algorithm takes as inputs the stream* S *and the secret key* k *and outputs the substring encrypted data structure* SES.
- $\mathsf{tk}_{T,S} \leftarrow \mathsf{SrchToken}(\mathsf{k}, T[1...m])$*: It is a probabilistic algorithm 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.
- $(s, \perp) \leftarrow \mathsf{Search}(\mathsf{tk}_{T,S}, \mathtt{SES})$*: It is a deterministic algorithm which takes as input a trapdoor* $\mathsf{tk}_{T,S}$ *and an substring encrypted structure* SES *and outputs the positions* s *in* S *that substring* T *occurs, or* $\perp$ *otherwise.*

A $\mathsf{S^3E}$ is correct if $\forall\lambda \in \mathbb{N}, \forall S \in \Sigma, \forall \mathsf{k} \leftarrow \mathsf{KeyGen}(1^\lambda), \forall \mathtt{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S), \forall \mathsf{tk}_{T,S} \leftarrow \mathsf{SrchToken}(\mathsf{k}, T[1...m]), \mathsf{Search}(\mathsf{tk}_{T,S}, \mathtt{SES})$ always returns the correct positions s in the string S or $\perp$ otherwise.

2.2 Security Model

Intuitively the security guarantee we ask for is **1**) given a probabilistic polynomial time adversary $\mathcal{A}$ with access to a substring encrypted structure SES, $\mathcal{A}$ cannot gain more partial information about the underlying stream of data S and **2**) given a set of trapdoor tokens for an adaptive generated set of queries $\mathbf{q} = (q_1, q_2, q_3, \ldots, q_o)$ associated with set of tokens $\mathbf{t} = (\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3, \ldots, \mathsf{tk}_o)$ $\mathcal{A}$ cannot learn anything for $\mathbf{q}$ and $\mathbf{t}$. Following the symmetric searchable paradigm we know it is impossible to achieve those two security guarantees without leaking some extra information as the observed in [7, 8, 12].

We express the security guarantees of the protocol in terms of the consolidated simutability [30]. First a leakage function $\mathcal{L}$ is defined, which expresses the leakage of a $\mathsf{S^3E}$ scheme to an adversary $\mathcal{A}$, though the transcripts of the protocol. The simulation framework assumes two games. The $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ game, in which adversaries can corrupt the parties they want and the $\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ one in which there is only benign behavior of each party. The security analysis narrows down to the design of a simulator $\mathcal{S}$, who tries to simulate the malicious behavior in the $\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ game, only through access to the leakage function $\mathcal{L}$. We say that a protocol is secure if $\mathcal{S}$ simulates indistinguishable views of the adversary $\mathcal{A}$ in the $\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ game.

Definition 2. *A leakage function* $\mathcal{L}$ *for a* $\mathsf{S^3E}$ *scheme comprises the following three leakage functions:*

- PreProcess *Leakage:* $\mathcal{L}_1$ *includes the padded size of the data stream* $n' >= |S|$.
- SrchToken *Leakage: The* SrchToken *Leakage* $\mathcal{L}_2$ *reveals the length of the token* $|\mathsf{tk}|$ *and how many common characters reside in it.*
- Search *Leakage:* $\mathcal{L}_3$ *leaks the how many times a substring token* tk *exists in the padded string* S *with dummy blocks.*

The adversary $\mathcal{A}$ plays the role of a semi-honest cloud and during the two games we assume a challenger $\mathcal{C}$ who interacts with $\mathcal{A}$. We describe the two games in algorithmic details in what is follows: $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ game:

- $\mathcal{C}$ runs $\mathsf{KeyGen}(1^\lambda)$ to obtain k.
- $\mathcal{A}$ chooses a string $S \in \Sigma$, sends it to $\mathcal{C}$ and $\mathcal{C}$ replies with $\mathtt{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S)$ to $\mathcal{A}$.
- $\mathcal{A}$ issues a polynomial number of adaptively chosen queries $\mathbf{q} = (q_1, q_2, q_3, \ldots, q_o)$ and receives from $\mathcal{C}$ a set of tokens $\mathbf{t} = (\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3, \ldots, \mathsf{tk}_o)$.
- Finally $\mathcal{A}$ outputs $v = (\mathtt{SES}, \mathbf{t})$.

$\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ game:

- $\mathcal{A}$ outputs a string stream S.
- The simulator $\mathcal{S}$ through the leakage $\mathcal{L}$ generates SES and forwards it to $\mathcal{A}$.
- $\mathcal{A}$ issues a polynomial number of queries $\mathbf{q} = (q_1, q_2, q_3, \ldots, q_o)$. $\mathcal{S}$ replies to each of the queries through the leakage function $\mathcal{L}$ with $\mathbf{t} = (\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3, \ldots, \mathsf{tk}_o)$.
- Finally $\mathcal{A}$ outputs $v = (\mathtt{SES}, \mathbf{t})$.

Definition 3. *A* $\mathsf{S^3E}$ *scheme is adaptively* $\mathcal{L}$*-semantically secure against a probabilistic polynomial time adversary* A *if there is exists a polynomial Simulator* $\mathcal{S}$ *such that for all polynomial time distinguishers* $\mathcal{D}$*:*

$$|\Pr[\mathcal{D}(v) = 1 : v \leftarrow \mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}] -$$
$$\Pr[\mathcal{D}(v) = 1 : v \leftarrow \mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}]| \leq \mathsf{neg}(\lambda)$$

3 Related work

The ORAM paradigm [22] 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 [22] the bandwidth overhead is polylogarithmic, which has been reduced down to logarithmic in subsequent work [15, 35, 38, 40, 41]. 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 [6–8, 12, 37], 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 the 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 [9], [17], [16]. Here we present a detailed analysis of the state of the art in substring searchable encryption protocols. Chase *et al.* [9] 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 is $O(n)$ which hinders a big constant factor, which can goes up to 20 [1, 5, 23, 31]. In [16] the authors extended the efficient SSE scheme for boolean queries from [6] in order to support substring matching. The idea is to build an index of overlapping k-grams, to prepend 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 [17] 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 contradicts the security definition of subset sum problem, which asks for impossibility of an adversary to find a solution to a specific instance. More specifically the user uploads a special instance $\mathbf{T}$ of a subset sum problem such that given a trapdoor R_i associated with a substring, the solution can be solved in time O(m) by the cloud; with the special property that the integers which sum to $\mathbf{T}$, parametrized by R_i are the positions of the substring in the string. 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 [34] 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.

Papadopoulos *et al.* [36] addressed the problem of authenticating substring queries without privacy and various work for pattern matching adopts the two party computation model [13, 19, 24, 25] 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 General Idea

In order to reduce the storage cost for our Substring Searchable Symmetric encryption scheme ($\mathsf{S^3E}$) we first substitute the storage expensive suffix tree of the state of art work in [9] 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 $O(m + logn)$. Let us now consider a concrete example to uncover its details. Suppose S=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, lakis, 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 for the cloud to retrieve the right encrypted index position from `SA` ought to 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.* [20], 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, plus there should be one extra round of communication due to the ORAM protocol in order to rebuild the specific path.

In this paper we are taking a different approach, which achieves storage efficiency. The problem arises from the fact that the server does not hold the original string in order to perform a binary search, which is indexed by the suffix array `SA`. We take advantage of the *self-indexed* data structure Ferragina-Manzini index, called hereafter FM index [18]. 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) [5]. 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 ($\mathsf{S^3E}$) 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.

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. However, for the construction of S^3E we only need a compressed version of the intermediate steps and not the final string W. 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 shorted lexicographically in an ascending way. A real world example is shown in figure 2 for string $S = lalakis$. The upper table of the figure shows the permutations and the final shorted matrix W is shown at the bottom matrix. The transformation is the first step towards compression with the LF mapping that is shown below.

Algorithm 1: BWT transformation

Input: String S
Output: BWT(S) = W
l=length(S)+1;
S.append(\$);
i=0;
while i!l **do**
  r_i=rotate(s,\$) // *The rotate algorithm permutes the characters of the original string and returns the permuted string*;
  W.addrow(r_i) // *It adds the permuted row from the previous step to the matrix W*;
  $i++$;
end
return Sorted.W;

l	a	l	a	k	i	s	$	$	l	a	l	a	k	i	s
a	l	a	k	i	s	$	l	a	k	i	s	$	l	a	l
l	a	k	i	s	$	l	a	a	l	a	k	i	s	$	l
a	k	i	s	$	l	a	l	i	s	$	l	a	l	a	k
k	i	s	$	l	a	l	a	k	i	s	$	l	a	l	a
i	s	$	l	a	l	a	k	l	a	k	i	s	$	l	a
s	$	l	a	l	a	k	i	l	a	l	a	k	i	s	$
$	l	a	l	a	k	i	s	s	$	l	a	l	a	k	i

Fig. 2: BWT Transformation. The upper table shows the cyclic permutations for the string S=lalakis. As a first step for the construction of the BWT matrix the end symbol \$ is appended at the end of S, which lexicographically precedes all alphabetical symbols. Then at each row a permutation of \$ around the string is shown. The result is matrix BWM[length(S)+1][length(S)+1]. As a second step the rows are sorted lexicographically at the second bottom table of the figure starting from the first character of each string. The light gray first column shows the order of the characters after sorting and the last column is the result of the transformation BWT(S)=sllkaa\$s.

LF Mapping The LF Mapping technique takes the first F and last L columns from the BWT transformation and through an iterative process as described in 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 3. At the first iteration the pointer indicates the first position in both columns F, L. For the next iteration the L character 's' indicates the index for the first column F, which can be found at its last position with F[7] =s. The current character at the L column is appended to a stack D. For a 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 2: LF Mapping

Input: First (F), Last column (L) from BWT
Output: S
D=0 // Initialize the stack D;
l=length.(F) // the length of F equals the length of L;
i=0;
while $L[i]! = \$$ **do**
  D.push(L[i]);
  i=find.F[L[i]]// find.[] denotes the index number in array [] that the element is. For instance
   find.F['s']=7 ;
while $D! = \backslash'0'$ **do**
  S=S+D.pop;
return S;

F	L	F	L	F	L	F	L	F	L	F	L	F	L	F	L
$	s	$	s	$	s	$	s	$	s	$	s	$	s	$	s
a	l	a	l	a	l	a	l	a	l	a	l	a	l	a	l
a	l	a	l	a	l	a	l	a	l	a	l	a	l	a	l
i	k	i	k	i	k	i	k	i	k	i	k	i	k	i	k
k	a	k	a	k	a	k	a	k	a	k	a	k	a	k	a
l	a	l	a	l	a	l	a	l	a	l	a	l	a	l	a
l	$	l	$	l	$	l	$	l	$	l	$	l	$	l	$
s	i	s	i	s	i	s	i	s	i	s	i	s	i	s	i

Fig. 3: 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 ith index in the F column, which in turn maps to the ith entry in the L column. The entire procedure halts when L[i]==\$

FM Index Suffix array vanilla construction has $O(n^2 logn)$ asymptotic computational cost. This stems from the fact that we need to first sort the n suffices by performing $O(nlogn)$ comparisons and each comparison has cost n. Linear time algorithms have be achieved by first constructing a suffix tree and then traversing with a depth first edges with lexicographical order. However, our goal is to be storage efficient, meaning we want to eliminate the storage cost of a suffix tree which practically approximates a constant factor of $20n$ [1, 5, 23, 31]. We pick up the *skew* algorithm [27] 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 suffices: $\mathsf{pos} \bmod h, h \in \{1, 2, 3\}$ 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 $\mathsf{BWT}(S)$ 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. Notice that the F column is equivalent with the same range first letter suffices of the corresponding SA. $\mathtt{L} = \mathsf{BWT}(S)$ can be computed with the formula $\mathsf{BWT}(S)[i] = S[\mathtt{SA}[i] - 1]$ 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 $r_{\mathtt{F}}, r_{\mathtt{L}}$ accordingly. Finally $\mathtt{FM} = \{\mathtt{F}[i], \mathtt{L}[i], r_{\mathtt{F}}[i], r_{\mathtt{L}}[i], \mathtt{SA}[i]\}_{i=1}^{n}$

5 Intuition

In this section we provide some intuition about $\mathsf{S^3E}$ protocol before delving into its precise description in the follow up section. First we start with how the client encrypts the index based on the suffix array and the FM index described before hand and then we depict the overall substring query protocol. The notation used for the protocol is given in table 1.

σ	Vocabulary size
Σ	Vocabulary
S	Original stream
T	Substring query
n	Size of S
m	Size of T
SA	Suffix array
F	First Column of LF mapping
L	Last Column of LF mapping
LLSet	Hash map
LL	Linked List
$c_{\mathtt{F}_j}$	j^{th} character from F column
$c_{\mathtt{L}_j}$	j^{th} character from L column
r_{c_i}	Ranking of i^{th} character in the string S
$r_{\mathtt{F}_j}$	Ranking of j^{th} character from F column
$r_{\mathtt{L}_j}$	Ranking ofj^{th} character from L column
c^w	w^{th} character from the alphabet, $(1 \leq w \leq \sigma)$
λ	Security parameter
$\mathsf{F}(\cdot)$	Pseudorandom function (PRF)
$\Pi(\cdot)$	Psudorandom permutation (PRP)
$\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$	Symmetric encryption
$\mathsf{k_f}$	PRF key
$\mathsf{k_l}$	PRF key
k_{π_1}	PRP key
k_{π_2}	PRP key

Table 1: Notations

Concept. For our $\mathsf{S^3E}$ protocol we use a hash map LLSet, which is a set of tuples $\langle k, v\rangle$, with keys k to access values v. LLSet values entail the addresses for a set of linked lists $\{\mathtt{LL}_{c^1}, \mathtt{LL}_{c^2}, \mathtt{LL}_{c^3}, \cdots, \mathtt{LL}_{c^\sigma}\}$ and an array FM, which stores information about the position of substring in the string S. For the security of the scheme the user employs lightweight cryptographic primitives: a pseudorandom function $\mathsf{F}(\cdot)$, a pseudorandom permutation $\Pi(\cdot)$ respectively and a symmetric encryption scheme $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$.

The user computes a hash table of linked lists LLSet, where each position $\mathtt{LLSet}[\mathsf{F}_{\mathsf{k_f}}(c^w)], 1 \leq w \leq \sigma$ maps to the linked list $\mathtt{LL}_{c^w}$. The number of linked lists equals the number of distinct elements c, denoted as σ in the data stream S, where each symbol $c^w, 1 \leq w \leq \sigma$ comes from an alphabet Σ. The hash table is used to fetch all the positions of a character in the stream S from the linked lists $\mathtt{LL}_{c^w}$. Each linked list $\mathtt{LL}_{c^w}$ stores information concerning the retrieval of the position of c from S. More specifically each node in the list stores the following tuple: $\langle \mathtt{nptr}, \mathtt{addr}\rangle$, 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 FM index consists of three arrays, which keeps track of the F, L columns and the encrypted SA suffix array with positions of substrings and not all the suffixes as in [34]. User encrypts the FM index using $\mathsf{F}(\cdot)$, $\Pi(\cdot)$ and $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$. Finally the cloud traverses the FM index as in algorithm 2 and transmits the result to the user. 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 4). To recap, the user computes the suffix-array SA and the F, L columns through the BWT transformation.

FM Index Encryption (figure 4). The crux of the design is on how to allow fast indexing through a hash table, which means that the 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. This coupling makes a unique bit-stream which can be given as input to the PRF $\mathsf{F}_{\mathsf{k_f}}(\cdot)$ and serves as an index to the FM design. 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 $\mathtt{LL}_c$ for each character. Each node in the linked list maps to the address of the character c in the FM index, and has a pointer to the next same character node. The first node of each linked list is stored in

the LLSet hash table. In order to prevent frequency attacks we encrypt each key $\mathsf{F}_{\mathsf{k_f}}(c^w)$ in the LLSet hash map with another key $\mathsf{k_l}$ as follows: $\mathsf{F}_{\mathsf{k_f}}(c^w) \oplus \mathsf{F}_{\mathsf{k_l}}(c^w)$. Thus the cloud cannot perform a frequency attack offline without observing any token. The key of the hash map LLSet at $\mathsf{F}_{\mathsf{k_f}}(c^w) \oplus \mathsf{F}_{\mathsf{k_l}}(c^w)$ maps to the first element of the linked list $\mathtt{LL}_c$, which is encrypted as $\langle \mathtt{nptr}, \mathtt{addr} \rangle \oplus \mathsf{F}_{\mathsf{k_l}}(c^w)$[1]. As such, the frequency of each character before a search query is hidden. 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 $\mathsf{tk}_{T,S}$, 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 $\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{F}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_j}||c_{\mathtt{F}_j})$ and the value is $\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{L}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_j}||c_{\mathtt{F}_j})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{L}c_j}||c_{\mathtt{L}_j}), \mathsf{Enc}(\mathsf{pos}_j)$. Finally the user permutes all the tuples with a secure permutation: $\Pi_{\mathsf{k}_\pi}(t_j)$.

Search. During the Search phase on a substring query $\mathsf{tk}_{T,S} = \mathsf{F}_{\mathsf{k_f}}(T[1...m]) = \mathsf{F}_{\mathsf{k_f}}(T[1]), \mathsf{F}_{\mathsf{k_f}}(T[2]), \ldots \mathsf{F}_{\mathsf{k_f}}(T[m]), \mathsf{F}_{\mathsf{k_l}}(T[m])$ the cloud proceeds as follows: From the LLSet hash table it looks for the value with key $\mathsf{F}_{\mathsf{k_f}}(T[m]) \oplus \mathsf{F}_{\mathsf{k_l}}(T[m])$. This value maps to a linked list $\mathtt{LL}_c$, in which each node maps to the encrypted FM tuple $t_j = t_j^0, t_j^1, t_j^2 =$

$$\langle \underbrace{\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{F}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_j}||c_{\mathtt{F}_j})}_{\mathtt{F}},$$

$$\underbrace{\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{L}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_j}||c_{\mathtt{F}_j})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{L}c_j}||c_{\mathtt{L}_j})}_{\mathtt{L}}, \underbrace{\mathsf{Enc}(\mathsf{pos}_j)}_{\mathtt{SA}} \rangle_{j=1}^{n}.$$

In order to decrypt the first element of the linked list the cloud uses $\mathsf{F}_{\mathsf{k_l}}(T[m])$ as a key to decrypt $\langle \mathtt{nptr}, \mathtt{addr} \rangle \oplus \mathsf{F}_{\mathsf{k_l}}(c^w)$, in order to learn $\langle \mathtt{nptr}, \mathtt{addr} \rangle$. The cloud uses $\mathsf{F}_{\mathsf{k_f}}(T[m])$ and applies a xor operation on the F column at the ranges that it retrieved from the linked list of the c_m character $\mathtt{LL}_c$ and learns $\mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_j}||c_{\mathtt{F}_j})$. Afterwards it uses $\mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_j}||c_{\mathtt{F}_j})$ as a key to decrypt the first part of the L column element $\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{L}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_j}||c_{\mathtt{F}_j})$ and reveals $\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{L}_j})$. It then fetches the encrypted L column as $k = \mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{L}_j}), b = \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{L}_j}||c_{\mathtt{L}_j})$ in which $\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{L}_j}) = \mathsf{F}_{\mathsf{k_f}}(T[m-1])$ 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 $\mathsf{F}_{\mathsf{k_f}}(T[1])$. At this point the cloud returns to the user all the encrypted $\mathsf{Enc}(\mathsf{pos}_j)$ for the substrings. The user decrypts accepts the result as long as the decrypted position is in the range of the size of original stream without padding.

Second round. From the per-character one way function $\mathsf{F}_{\mathsf{k_f}}$ evaluation of the substring query: $\mathsf{tk}_{T,S} = C[1], C[2], \cdots, C[m] \leftarrow \mathsf{F}_{\mathsf{k_f}}(T[1...m])$ and the LF mapping the protocol leaks to the cloud in cleartext the exact differences of the positions of two encrypted substring in the stream S. 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 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: $\Pi_{\mathsf{k}_{\pi_1}}(t_j^0, t_j^1) = \pi_j^{0,1}, \Pi_{\mathsf{k}_{\pi_2}}(t_j^2) = \pi_j^2$. As such after the permutation $\underbrace{\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{F}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_j}||c_{\mathtt{F}_j})}_{\mathtt{F}}, \underbrace{\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{L}_j}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_j}||c_{\mathtt{F}_j})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{L}_j}||c_{\mathtt{L}_j})}_{\mathtt{L}}$ are stored in position $\pi_j^{0,1}$ at the FM array and $\underbrace{\mathsf{Enc}(\mathsf{pos}_j)}_{\mathtt{SA}}$ at position π_j^2. The cloud as traverses the token returns the permuted encrypted position of the substring token, the client applies the inverse permutation and fetches the correct cell from the FM array. By doing so the cloud cannot learn on its own, the encrypted position of a substring. The second permutation prevents him to leak this information as it needs the contribution of the user.

Dummy blocks. In order to obfuscate frequency analysis on the encrypted index from substring search queries we pad the data stream with dummy blocks. the core idea for padding is to produce dummy blocks from the vocabulary Σ depending on proportional to the ranking of the most frequent character. E.g: Original stream=abbcd and Σ=abcd, then the dummy blocks equal dc={ a, c, d}. Finally the user chooses uniformly at random $dpos \xleftarrow{\$} \{|dc|\}$ and appends the original string S at position $dpos$ of dc.

[1] Notice that even if we use a one time pad with the same key for two different elements: $a = \langle \mathtt{nptr}, \mathtt{addr} \rangle, b = \mathsf{F}_{\mathsf{k_f}}(c^w)$ an adversary by xoring the two ciphertexts encrypted under the same key $\mathsf{F}_{\mathsf{k_l}}(c^w)$, learns $ab = \langle \mathtt{nptr}, \mathtt{addr} \rangle \oplus \mathsf{F}_{\mathsf{k_f}}(c^w)$, which is a one time pad encryption of $\langle \mathtt{nptr}, \mathtt{addr} \rangle$ with key $\mathsf{F}_{\mathsf{k_f}}(c^w)$.

Following the previous example; if dpos=1 then S'=aabbdccd. The user proceeds with the construction of the FM index and its encryption as previously described. The cloud responds in the finally round with the encrypted position *pos*. User accepts the result as correct if $dpos \leq pos \leq n-dpos$ and $pos+m < dpos+n$

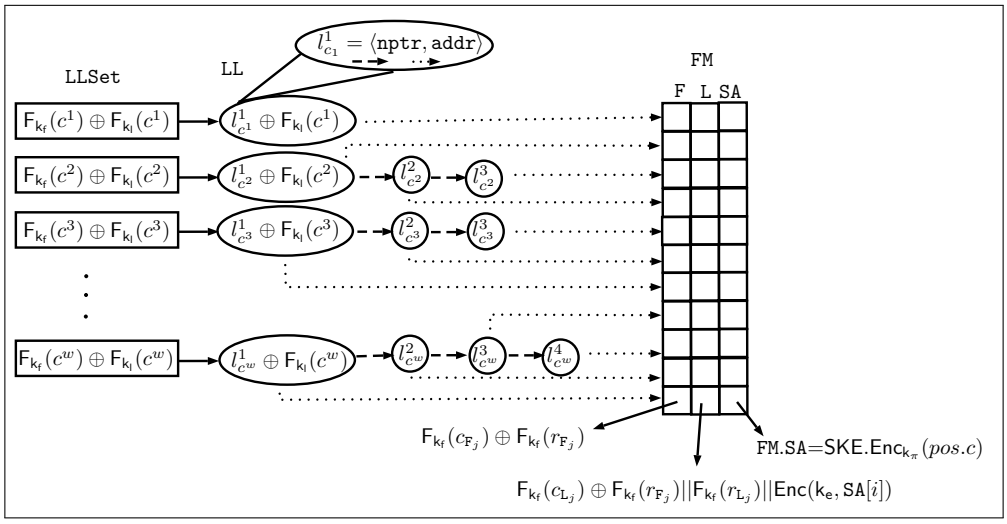

Fig. 4: The encrypted FM index construction.

6 Protocol

6.1 $\mathsf{S^3E}$ Description

We are now ready to give the full details of our substring searchable symmetric encryption protocol:

- $\mathsf{k} \leftarrow \mathsf{KeyGen}(1^\lambda)$: This algorithm run by the user takes as input the security parameter 1^λ and generates random keys $\mathsf{k} = (\mathsf{k_f}, \mathsf{k_l}, \mathsf{k}_{\pi_1}, \mathsf{k}_{\pi_2}, \mathsf{k_e})$ for a PRF $\mathsf{F}_{\mathsf{k_f}} : \{0,1\}^\lambda \times \{0,1\}^\nu \rightarrow \{0,1\}^\mu$, a PRP $\Pi_{\mathsf{k}_\pi} : \{0,1\}^\lambda \times \{0,1\}^\nu \rightarrow \{0,1\}^\nu$ and a symmetric encryption algorithm $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$. Finally it outputs k to the user. For the generation of the keys we assume a source of randomness $\mathcal{R}$ and a pseudorandom generator G seeded with $s_\mathsf{f} \xleftarrow{\$} \mathcal{R}, s_\mathsf{l} \xleftarrow{\$} \mathcal{R}, s_{\pi_1} \xleftarrow{\$} \mathcal{R}, s_{\pi_2} \xleftarrow{\$} \mathcal{R}, s_\mathsf{e} \xleftarrow{\$} \mathcal{R}$: $(\mathsf{k_f}, \mathsf{k_l}, \mathsf{k}_{\pi_1}, \mathsf{k}_{\pi_2}, \mathsf{k_e}) \leftarrow \mathsf{G}(s_\mathsf{f}), \mathsf{G}(s_\mathsf{l}), \mathsf{G}(s_{\pi_1}), \mathsf{G}(s_{\pi_2}), \mathsf{G}(s_\mathsf{e})$.
- $\mathtt{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S)$: User owns a stream S, which contains characters $c \in \Sigma$. S has n characters in total and k distinct elements. User:
  1. Let max_c be the cardinality of most frequent character and f_{c_i} the frequency of character c_i. User chooses dummy characters $dc = \sum_{j=1}^{k} \mathsf{max}_c - f_{c_j}$ that constitute a dummy stream. The user chooses uniformly at random $dpos \xleftarrow{\$} \{|dc|\}$ and appends the original string S at position *dpos* of dc
  2. Computes the suffix array $\mathtt{SA}$ and the $\mathtt{F}, \mathtt{L}$ columns and stores it as the $\mathtt{FM}$ index: $\mathtt{FM} = \mathtt{F}||\mathtt{L}, \mathtt{SA}$.
  3. Encrypts its element of $\mathtt{SA}$ array with $\mathsf{SKE.Enc}(\mathsf{k_e}, \mathtt{SA}[i]), 1 \leq i \leq n$.
  4. Applies the PRF to each element of $\mathtt{F}$ as follows: $\mathtt{F}[i] = \mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{F}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_i}||c_{\mathtt{F}_i})||\mathsf{Enc}(\mathsf{k_e}, \mathtt{SA}[i])$.
  5. Computes $\mathtt{L}[i] = \mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{L}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_i}||c_{\mathtt{F}_i})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{L}_i}||c_{\mathtt{L}_i})$.
  6. Applies a pseudorandom permutation Π_{k_π} to the tuples $t_i = \langle \mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{F}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_i}||c_{\mathtt{F}_i})\mathsf{Enc}(\mathsf{k_e}, \mathtt{SA}[i]), \mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{L}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_i}||c_{\mathtt{F}_i})||\mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{L}_i}||c_{\mathtt{L}_i})$ using π_1 and with π_1 user permutes $\mathsf{Enc}(\mathsf{k_e}, \mathtt{SA}[i])\rangle_{j=1}^{n} : \Pi_{\mathsf{k}_{\pi_1}}(t_i^0, t_i^1) = \pi_i^{0,1}, \Pi_{\mathsf{k}_{\pi_2}}(t_i^2) = \pi_i^2 = \mathtt{FM}'$.
  7. For every distinct character in $\mathtt{F}[i] = \mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{F}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\mathtt{F}_i}||c_{\mathtt{F}_i})$ the user builds a linked list $\mathtt{LL}_c$ and each node stores $\mathtt{LL}_c.\mathtt{nptr}$ for the next node of the list and $\mathtt{LL}_c.\mathtt{addr}$ which points to the tuple t_i with a matching $\mathsf{F}_{\mathsf{k_f}}(c_{\mathtt{F}_i})$. Finally it encrypts the first element of each linked list $\mathtt{LL}_c$ with $\mathsf{F}_{\mathsf{k_l}}(c_i) : \langle \mathtt{nptr}, \mathtt{addr} \rangle \oplus \mathsf{F}_{\mathsf{k_l}}(c_i)$.

8. The head pointers of the collections of all linked lists are stored in a hash table $\texttt{LLSet}$ with key $k = \mathsf{F}_{\mathsf{k_f}}(c_i) \oplus \mathsf{F}_{\mathsf{k_l}}(c_i)$ and value v a pointer to the head of the list $\texttt{LL}_c$, which stores information about the $\mathsf{F}_{\mathsf{k_f}}(c_i)$ character, meaning all its positions to the encrypted $\texttt{FM}$ index.
9. Finally outputs $\texttt{SES} = (\texttt{LLSet}, \texttt{LL}_c, \texttt{FM}')$ and keeps only the keys $\mathsf{k} = (\mathsf{k_f}, \mathsf{k_l}, \mathsf{k}_{\pi_1}, \mathsf{k}_{\pi_2}, \mathsf{k_e})$.

– $\mathsf{tk}_{T,S} \leftarrow \mathsf{SrchToken}(\mathsf{k}, T[1...m])$: 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 $\texttt{SES}$. User with his secret PRF key $\mathsf{k_f}$ computes $\mathsf{tk}_{T,S} = C[1], C[2], \cdots, C[m] \leftarrow \mathsf{F}_{\mathsf{k_f}}(T[1...m]), \mathsf{F}_r(T[m])$ and forwards $\mathsf{tk}_{T,S}$ to the cloud.
– $(s, \perp) \leftarrow \mathsf{Search}(\mathsf{tk}_{T,S}, \texttt{SES})$: The cloud parses the token query $\mathsf{tk}_{T,S} = C[1], C[2], \cdots, C[m]$ and searches the position in S from the encrypted index $\texttt{SES}$ as follows:
1. $u = find(\texttt{LLSet}, C[m])$*//find in dictionary* $\texttt{LLSet}$ *the value u with key* $C[m]$*.u is a pointer to the head of a list* $\texttt{LL}$*, which stores pointers to all characters* $T[m]$ *in* S
2. **if** $u == \perp$ **return** $\perp$
3. **while** $u \neq \perp$ **do**
   $K = K \cup u.\texttt{addr}$ *//traverse the list and store in the set* K *the addresses of the characters.*
   $u = u.\texttt{nptr}$
4. **for** $p = m-1; p > 1; p = p-2$
   for $i = 1; i < size(K); i++$
   if $\texttt{SES.L}[K[i])^{(1)} == C[p-1]$*//Store in the set* $KEYS$ *only the elements from the* $\texttt{F}$ *column, whose associated* $\texttt{L}$ *element equals the next character from* C *in a backword order.* $\texttt{SES.L}[K[i]]^{(1)}$ *maps to* $\mathsf{F}_{\mathsf{k_f}}(c_{\texttt{L}_{c_i}}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\texttt{F}_{c_i}} || c_{\texttt{F}_j})$ *and* $\texttt{SES.L}[K[i]]^{(2)}$ *to* $\mathsf{F}_{\mathsf{k_f}}(r_{\texttt{L}_i})$.
   $KEYS = KEYS \cup \texttt{SES.L}[K[i]]$
   else $K = K - K[i]$ *//Remove all the non matched elements from the key set* K.
   if $K == \perp$ **return** $\perp$
   for $i = 1; i < size(KEYS); i++$
   $r_{\texttt{F}_{c_i}} = C[p] \oplus KEYS^{(1)}[i]$
   $z = r_{\texttt{F}_{c_i}} \oplus KEYS^{(2)}[i]$ *//Compute the key from the* $\texttt{L}$ *column as* $KEYS^{(1)}[i] \oplus KEYS^{(2)}[i]$*, which corresponds to* $\mathsf{F}_{\mathsf{k_f}}(c_{\texttt{F}_i}) \oplus \mathsf{F}_{\mathsf{k_f}}(r_{\texttt{F}_i} || c_{\texttt{F}_j})$ *in the* $\texttt{F}$ *column of the* $\texttt{SES}$ *object.*
   if $\texttt{SES.F}[z] \neq \perp$
   continue
   else $KEYS = KEYS - KEYS[i]$ *//Remove all the non matched elements from the key set* $KEYS$.
   $K = KEYS$
5. **if** $K == \perp$ **return** $\perp$
6. Cloud sends to the user $\texttt{SES.FM}'[K]$
7. The client runs the inverse permutation Π_{π_2} to the K indexes and gets back $\{i'\}$ and asks the cloud for $\texttt{SES.FM}'[\{i'\}]$. After getting back the results the client decrypts $\mathsf{Enc}(\mathsf{k_e}, \texttt{SA}[\{i'\}])$ with $\mathsf{k_e}$ and learns the position *pos* of the asked substring T in S. User accepts the result as correct if $dpos \leq pos \leq n - dpos$ and $pos + m < dpos + n$

6.2 Security Analysis

We illustrate the security of the scheme pertaining to definition 3. More specifically we show the existence of a simulator $\mathcal{S}$ who has access to the leakage function $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$, and produces indistinguishable views to an adversary $\mathcal{A}$. Conceptually the proof demonstrates that an adversary $\mathcal{A}$, who can be a dishonest cloud cannot learn partially more information from what it can be leaked in an ideal work without malicious behaviors. Albeit the acceptable introduction of a leakage which is permitted in the proof, the technique has been broadly adopted in the queryable encryption schemes.

Theorem 1. *Let* $\mathsf{F}_{\mathsf{k_f}}, \Pi_{\mathsf{k}_\pi}, \mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ *be a pseudorandom function, a pseudorandom permutation, a semantically secure symmetric encryption scheme respectively, then our substring searchable symmetric encryption scheme* $\mathsf{S^3E}$ *is adaptively* $\mathcal{L}$*-semantically secure.*

The proof is omitted to Appendix B section.

Protocol	Search	Index [PS—CS]		Query [FR—LR]		VLS	Rounds	SL
CS[9]	$O(m+k)$	$20n$	$4(n+\Sigma_{i=1}^{2n-(2+N)}\sigma - child(i))$	$\frac{m^2+\theta}{2\theta}$	m+k	✓	3	SP+QPP+IIP+LIP+DF
FJKNRS[16]	$O(n)$	-		m	0	✓	1	SP
FHV[17]	$O(n-m)$	-		m	0	✗	1	✗
$\mathsf{S^3E}$	$O(m+k)$	$4n$	$4(n+\sum_{j=1}^{k}\mathsf{max}_c - f_{c_j})$	m	1	✓	2	SP+QPP+IIP+DF

Table 2: Comparison of existing substring searchable encryption protocols. Index space is further categorized in plaintext space index storage space (PS) and ciphertext space (CS). The overhead of [16] and [17] 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 leakage: QPP: Query prefix pattern, IIP: Index intersection pattern, LIP: Leaf intersection pattern, DF: index differences.

6.3 Comparison

We perform a comparison of our $\mathsf{S^3E}$ with existing solutions (cf. table 2). 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 [9] here we further elaborate its cost analysis from table 2.

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 $O(m+k+\sum_{j=1}^{k}\mathsf{max}_c - f_{c_j})$, where max_c is the most frequent character and f_{c_j} the frequency of character c_j.

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 $4*4n = 16n$. For each edge, suffix trees allocate one pointer for the beginning position of the substring in the stream and one for the end position of the substring in the stream increasing the space cost to $24n + 4*2*2n = 40n$. The space cost of the solution based on suffix trees [9] can be further reduced to $20n$ by eliminating the need to store suffix links and parent pointers. However, the extra dummy blocks further augment the storage overhead. Assuming a suffix tree with N internal nodes of size, each node is further padded with dummy children nodes so as to each node has σ number of children, where σ is the size of the vocabulary. Furthermore, the internal nodes are padded with up to $2n-2$ nodes where n is the size of the string. Finally the size of the extra dummy nodes in [9] scheme is: $\Sigma_{i=1}^{2n-(2+N)}\sigma - child(i)$, where $child(i)$ equals the number of children for internal node i in the suffix tree. In contrast, in $\mathsf{S^3E}$ we replace the space expensive suffix trees with suffix arrays and as such the index space cost is reduced from $20n$ bytes to $4n$ bytes.

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 [9], the user encrypts $2n$ substrings which equal the number of edges of the suffix tree, n leaves and n characters of the original stream, resulting in $4n$ encryptions. In our solution thanks to the `FM` mapping the user does not need to send the original stream encrypted. It only 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 $4(n+\sum_{j=1}^{k}\mathsf{max}_c - f_{c_j})$ encrypted values to the cloud, in total.

Query. For the query size we assume a block cipher of size θ and a substring query of size m. In [9] 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 $1+2+\cdots+\frac{m}{\theta} = \frac{m^2+\theta}{2\theta}$ during the first round. At the last round the user asks for the positions of each character separately augmenting by a factor of m the substring size. In $\mathsf{S^3E}$ the substring query has only per character encryptions of each character in the first round plus a ciphertext fot 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 [9].
Rounds. For the rounds of communications $\mathsf{S^3E}$ can return the substring search results in 2 rounds of communication; independent from the size of the stream or the substring query. During the first round the client sends an encrypted substring query and the cloud responds with encrypted addresses of the corresponding suffix array positions. At the second round the client decrypts the permuted position of the suffix tree and queries the cloud for the unpermuted encrypted position.
Security Leakage. Concerning the search leakage, Chase *et al.* [9] scheme leaks the search pattern, meaning an untrusted could can 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 $\mathsf{S^3E}$ we avoid the use of a suffix tree, $\mathsf{S^3E}$ does not leak the leaf pattern. We inherit though, the index intersection pattern, which reveals similarities between the different characters of a token and the query prefix patter, which leaks similarities of the characters of a token. As in [9] our scheme revealsto the cloud differences of the indexes when a user asks for substrings that they do 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. $\mathsf{S^3E}$ 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. However as part of future work, malicious adversaries should be addressed in terms of a verifiable protocol with completeness and soundness.

7 Performance

In this section we present our implementation results. We demonstrate the practicality of $\mathsf{S^3E}$ with benchmark experiments, comparing our results with the scheme of Chase *et al.* [9], in order to validate the claims of our performance improvements. To accomplish the comparison we also implemented the suffix tree based construction of [9] called hereafter ST.

7.1 Implementation

For the benchmark experiments we used a machine running Ubuntu 14.04 with kernel version 3.19.0-29. The machine has 8GB RAM memory and is equipped with an INTEL Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz processor with 4 cores. We instantiated the PRF, PRP using `Poly1305` + `Salsa20`, `BLAKE2b` [2], which outperform their competitors in computational efficiency. The most computational heavy operation on the client side is the computation of the index, which becomes the bottleneck of the total performance. In our benchmarks, since the code will vary in different machines, we choose to isolate the code in different modules in order to evaluate their relevant performance. Thus, the first module consists of operations that take place in cleartext data, the second module entails the cryptographic primitives used to encrypt the `FM` index and the third module consists also of all `I/O` operations in order to serialize the encrypted `FM` index. For our benchmarks we used three synthetic datasets of $2, 4$ and 26 alphabet size of variable size: $10^2, \ldots, 10^6$ characters. We varied in either cases the size of the corpus in order to observe the feasibility results in these variations.

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 fare observations about the performance of the protocols on real metrics. In 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 **1.56** on average which approximates **1.8** when index is constructed on data sets with 10^6 elements. 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. The blowup on the computation time is not only due to the large number of extra dummy blocks but mostly by the way the encrypted suffix tree is encrypted in [9]: iterating the tree through its internal and leaf nodes, then for each node 2 PRF and 2 block cipher evaluations are required. In contrast in $\mathsf{S^3E}$ the iteration of the index comes per character in the suffix array, which asks for 1 PRF and 2 block cipher invocations. However, the per character encryption allows us to pre-compute the encrypted values by knowing only the alphabet, before learning the data stream, incurring very low storage overhead. In what it follows we present in details the benchmark results for index, substring query token and response computation costs of both schemes.

7.2 Benchmarks

Index We built synthetic data streams using three different vocabularies consisting of different sizes: 2,4 and 26. We started with small data streams , composed by few characters and we scaled up to data streams with millions of elements. To construct each synthetic data set we choose each character of the stream from a uniform distribution. In tables 3,4, we depict the storage overhead incurred by the computation of the encrypted index using the suffix array in $\mathsf{S^3E}$ and the suffix tree in ST scheme of [9].

#Vocabulary	**#Characters**				
	10^2	10^3	10^4	10^5	10^6
2	35KB	325KB	3.28MB	33.2MB	310MB
4	39KB	351KB	3.42MB	33.7MB	314MB
26	61KB	387KB	3.49MB	33.6MB	321MB

Table 3: $\mathsf{S^3E}$ Index overhead

#Vocabulary	**#Characters**				
	10^2	10^3	10^4	10^5	10^6
2	61KB	579KB	6.5MB	65MB	608MB
4	60KB	567KB	6.4MB	63MB	589MB
26	59KB	559KB	6.3MB	61MB	575MB

Table 4: [9] Index overhead

We observed an increased overhead in the size of the encrypted index for the ST scheme [9], compared with ours as expected. On average, over all the the data sizes, for different vocabularies the gain of $\mathsf{S^3E}$ over ST approximates a factor of **1.56**. 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 in storage device is not capable of computing the encrypted index if that comes in an increased communication overhead.

We also measured the computation time of the encrypted index in both schemes in tables 5, 6. The $\mathsf{S^3E}$ index construction time outperforms ST, abiding the increased size from the previous experiments. 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.

#Vocabulary	**#Characters**				
	10^2	10^3	10^4	10^5	10^6
2	8ms	71ms	79ms	8.57s	87.46s
4	8ms	81ms	79ms	8.58s	96.05s
26	8ms	111ms	79ms	9.02s	101.35s

Table 5: $\mathsf{S^3E}$ computation time

#Vocabulary	#Characters				
	10^2	10^3	10^4	10^5	10^6
2	41ms	342ms	4.61s	43.57s	356.32s
4	42ms	347ms	4.61s	44.25s	400.12s
26	44ms	359ms	4.71s	44.29s	451.23s

Table 6: [9] Index computation time

Query Encryption We run experiments in order to compute the storage overhead during the SrchToken phase. The token consists of a sequence of characters from different synthetic data sets of various sizes. We beheld a reasonable increase in both the size of the encrypted query (figures 5,6) and its computation time as the query size increases. $\mathsf{S^3E}$ 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 $\mathsf{S^3E}$ 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 $ct_1 = PRF_1(T[1]), k_1 = PRF_2(T[1]), ct_2 = PRF_1(T[1]T[2]), k_2 = PRF_2(T[1]T[2])$ and so on. Finally the client forwards to the cloud: $\{Enc_{k_i}(ct_i)\}_{i=1}^{m}$.

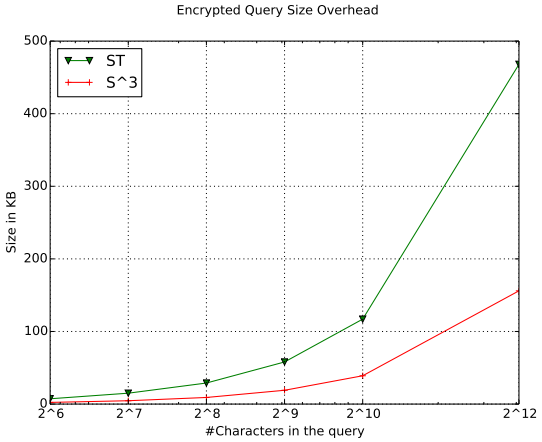

Fig. 5: Token storage overhead for both schemes

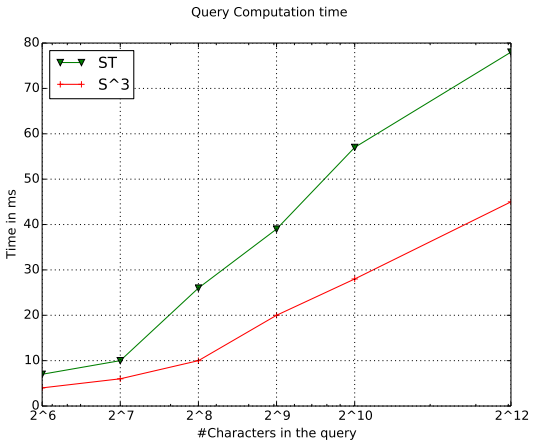

Fig. 6: Token computation time for both schemes

Response Overhead Finally, in figures 7, 8 we discern a slight outperformance of $\mathsf{S^3E}$ compared with ST in terms of substring response time. For the experiments we computed tokens of various lengths and perform a search on data streams of different sizes. 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, $\mathsf{S^3E}$ surmounts ST [9] for the computation of the response at the cloud side. This outperformance is due to the increased size of the encrypted index in [9] with dummy blocks, based on a suffix tree data structure.

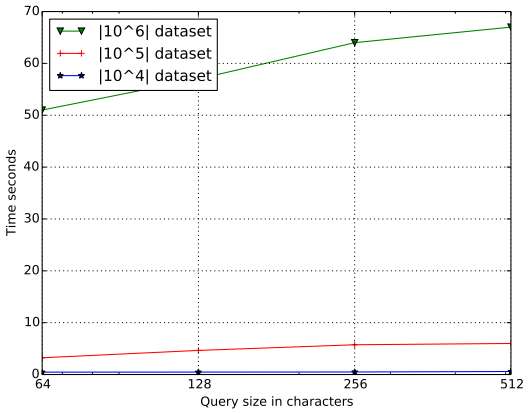

Fig. 7: Response time for $\mathsf{S^3E}$ scheme

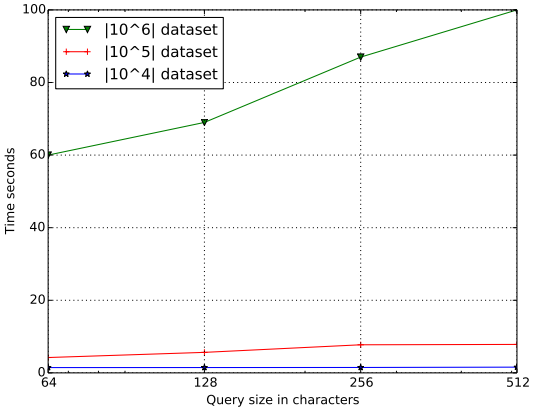

Fig. 8: Response time for ST [9] scheme

8 Conclusion

We designed and analyzed a substring searchable symmetric encryption protocol $\mathsf{S^3E}$, which achieves better storage performance and variable substring size than state of the art work [9]. The idea of our protocol is to leverage the self-indexing mechanism of `FM` index, which stores only n integer positions

of its substrings, without the need to store the mapping of substring in S. Our protocol is provably secure under the real-ideal indistinguishable simulation paradigm. We also implemented our protocol and compared it with the state of the art work [9], showing its notable performance improvement in terms of storage overhead and computation time.

Bibliography

[1] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *In Proceedings of the Ninth International Symposium on String Processing and Information Retrieval. Springer-Verlag, Lecture Notes in Computer Science*, 2002.

[2] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. Blake2: Simpler, smaller, fast as md5. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, pages 119–135, Berlin, Heidelberg, 2013. Springer-Verlag.

[3] Bitglass. `http://www.bitglass.com/company/news/press_releases/patentedencryption`.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, Oct. 1977.

[5] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[6] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 353–373, 2013.

[7] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Proceedings of the Third International Conference on Applied Cryptography and Network Security*, ACNS'05, pages 442–455, Berlin, Heidelberg, 2005. Springer-Verlag.

[8] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, pages 577–594, 2010.

[9] M. Chase and E. Shen. Substring-searchable symmetric encryption. Cryptology ePrint Archive, Report 2014/638, 2014. `http://eprint.iacr.org/2014/638`.

[10] Ciphercloud. `http://www.ciphercloud.com/technologies/encryption/`.

[11] Ciphercloud. Q2-global-cloud-data-security-report. `http://pages.ciphercloud.com/rs/830-ILB-474/images/Q2-Global-Cloud-Data-Security-Report.pdf`.

[12] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 79–88, New York, NY, USA, 2006. ACM.

[13] E. De Cristofaro, S. Faber, and G. Tsudik. Secure genomic testing with size- and position-hiding private substring matching. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 107–118, New York, NY, USA, 2013. ACM.

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

[15] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion oram: A constant bandwidth blowup oblivious ram. Cryptology ePrint Archive, Report 2015/005, 2015. `http://eprint.iacr.org/2015/005`.

[16] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, pages 123–145, 2015.

[17] S. Faust, C. Hazay, and D. Venturi. Outsourced pattern matching. In F. Fomin, R. Freivalds, M. Kwiatkowska, and D. Peleg, editors, *Automata, Languages, and Programming*, volume 7966 of *Lecture Notes in Computer Science*, pages 545–556. Springer Berlin Heidelberg, 2013.

[18] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[19] R. Gennaro, C. Hazay, and J. S. Sorensen. Automata evaluation and text search protocols with simulation-based security. *J. Cryptology*, 29(2):243–282, 2016.

[20] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, pages 1–18, 2013.

[21] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, Aug. 1986.
[22] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.
[23] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. Information retrieval. chapter New Indices for Text: PAT Trees and PAT Arrays, pages 66–82. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
[24] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *J. Cryptology*, 23(3):422–456, 2010.
[25] C. Hazay and T. Toft. Computationally secure pattern matching in the presence of malicious adversaries. In *Advances in Cryptology-ASIACRYPT 2010*, pages 195–212. Springer Berlin Heidelberg, 2010.
[26] Hitachi. Searchable encryption: A technology for supporting secure application. `http://www.hitachi.com/rd/portal/contents/story/searchable_encryption/index.html`.
[27] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, pages 943–955, 2003.
[28] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, Mar. 1987.
[29] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
[30] Y. Lindell. How to simulate it - A tutorial on the simulation proof technique. *IACR Cryptology ePrint Archive*, 2016:46, 2016.
[31] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.
[32] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, Apr. 1976.
[33] Mitsubishielectric. `http://www.mitsubishielectric.com/news/2013/0703.html`.
[34] T. Moataz and E.-O. Blass. Oblivious substring search with updates. Cryptology ePrint Archive, Report 2015/722, 2015. `http://eprint.iacr.org/2015/722`.
[35] T. Moataz, T. Mayberry, and E.-O. Blass. Constant communication oram with small blocksize. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 862–873, New York, NY, USA, 2015. ACM.
[36] D. Papadopoulos, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Practical authenticated pattern matching with optimal proof size. *Proc. VLDB Endow.*, 8(7):750–761, Feb. 2015.
[37] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.
[38] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o ((logn) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214. Springer Berlin Heidelberg, 2011.
[39] Skyhighnetworks. `https://www.skyhighnetworks.com/cloud-encryption/`.
[40] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267, May 2013.
[41] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.
[42] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.
[43] Virtualworks. `http://www.virtualworks.com/viaworks-enterprise-search/`.
[44] P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11, Oct 1973.

A Cryptographic Primitives

A.1 Pseudorandom functions (PRF)

Let the family of all functions in the universe from a domain X to a range Y to be $Func[X, Y]$. A truly random function $f \xleftarrow{\$} Func[X, Y]$ is chosen randomly from the set of $Func$. The set of all 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 $f_k : X \to Y$ as a function from the set of all functions from X to Y as soon as a particular key k is fixed.

Definition 4. *Let Func=$\{F : X \to Y\}$ be a function family for all functions F that map elements from the domain X to the range R. Then a PRF = $\{f_k : X' \to Y'\} \subseteq Func$ for $k \xleftarrow{\$} K$, 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[21]. Intuitively, an adversary $\mathcal{A}$ is given access to an oracle that on input x from a domain X, flips a coin $b \xleftarrow{\$} \{0, 1\}$ and if $b = 0$ then it outputs $y = f(x)$, for $f \in Func[X, Y]$, otherwise it outputs $y = f_k(x)$. $\mathcal{A}$ issues queries to the oracle polynomially many times on input of the security parameter λ. Finally $\mathcal{A}$ outputs a guess b' for the bit b.

The advantage of a probabilistic polynomial time algorithm $\mathcal{A}$ in the PRF game is

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

Definition 5. *A PRF is computationally secure if all probabilistic polynomially time algorithms $\mathcal{A}$ have advantage in the PRF game: $\frac{1}{2} + \epsilon(\lambda)$, for a negligible function ϵ on the security parameter λ.*

A.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 $\mathcal{A}$ in the PRP game is

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

Definition 6. *A PRP is computationally secure if all probabilistic polynomially time algorithms $\mathcal{A}$ have advantage in the PRP game: $\frac{1}{2} + \epsilon(\lambda)$, for a negligible function ϵ on the security parameter λ.*

A.3 Symmetric Key Encryption

A symmetric key encryption scheme $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ consists of three algorithms. Gen takes as input a security parameter λ and outputs the secret key sk. The probabilistic encryption algorithm Enc takes as input the secret key sk and a plaintext x form the plaintext space $\mathcal{P}$ and outputs the ciphertext c. The decryption algorithm $\mathsf{SKE.Dec}$ takes as input a ciphertect form the ciphertext space $\mathcal{C}$ and the secret decryption key sk and outputs the plaintext $x \in \mathcal{P}$. Correctness follows $\iff \forall \mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda), \mathsf{SKE.Dec}((\mathsf{Enc}(\mathsf{sk}, x))) = x, \forall x \in \mathcal{P}$. Security is modeled with the standard game based indistinguishability experiment for polynomial probabilistic time adversary $\mathcal{A}$.

$\underline{\mathsf{PrivK}_{\mathcal{A},\mathsf{SKE}}(\lambda)}$:

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

If $b' = b$ $\mathcal{A}$ succeeds and the experiment $\mathsf{PrivK}_{\mathcal{A},\mathsf{SKE}}(\lambda) = 1$.

Definition 7. *A symmetric encryption scheme SEK has indistinguishable encryptions if the probabilities $\Pr[\mathsf{PrivK}_{\mathcal{A},\mathsf{SKE}}(\lambda) = 1] \leqslant \frac{1}{2} + \mathsf{neg}(\lambda)$.*

Game	Change	Indistinguishability Argument
Game_0	$\mathsf{Game}_0 = \mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$	By definition
Game_1	Replace $\mathsf{F}_{\mathsf{k_f}}, \mathsf{F}_{\mathsf{k_l}}$	Pseudorandomness of $\mathsf{F}_{\mathsf{k_f}}$
Game_2	Replace $\Pi_{\mathsf{k}_{\pi_1},\pi_2}$	Pseudorandomness of Π_{k_π}
Game_3	Replace $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$	Semantically secure $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$
Game_4	$\mathsf{Game4} = \mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$	By definition

Table 7: Hybrid games

B Security Analysis

Theorem 2. *Let* $\mathsf{F}_{\mathsf{k_f}}, \Pi_{\mathsf{k}_\pi}, \mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ *be a pseudorandom function, a pseudorandom permutation and a semantically secure symmetric encryption scheme respectively, then our substring searchable symmetric encryption scheme* $\mathsf{S^3E}$ *is adaptively* $\mathcal{L}$*-semantically secure.*

Proof. In the $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ game the adversary plays the role of the cloud and the challenger the role of the client. In the beginning the Challenger selects uniformly at random keys $\mathsf{k} = (\mathsf{k_f}, \mathsf{k_l}, \mathsf{k}_{\pi_1}, \mathsf{k}_{\pi_2}, \mathsf{k_e})$ for a PRF $\mathsf{F}_{\mathsf{k}_f,r} : \{0,1\}^\lambda \times \{0,1\}^\nu \to \{0,1\}^\mu$, a PRP $\Pi_{\mathsf{k}_\pi} : \{0,1\}^\lambda \times \{0,1\}^\nu \to \{0,1\}^\nu$ and a symmetric encryption algorithm $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$. Upon receipt of a stream S of size n, the Challenger employs the $\mathtt{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, S)$ as presented in section 6 and forwards $\mathtt{SES}$ to $\mathcal{A}$. We distinguish between matching q^m and non-matching queries $q^{nm} : \mathbf{q} = \bigcup q^{nm} q^m$. Upon receiving the substring queries $\mathbf{q}$, the Challenger with $\mathsf{F}_{\mathsf{k_f}}$ computes $\mathsf{tk}_{T,S} = C[1], C[2], \cdots, C[m] \leftarrow \mathsf{F}_{\mathsf{k_f}}(T[1...m]), \mathsf{F}_r(T[m])$. We assume for the ease of readability that adversary issues only matching queries q^m. Finally $\mathcal{A}$ receives $\mathbf{t} = (\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3, \ldots, \mathsf{tk}_o)$ for each substring query.

Within a sequence of hybrid games we show the indistinguishable transformation of $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ game to eventually the $\mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ game, which concludes the proof. The simulator $\mathcal{S}$ computes the simulated encrypted index $\mathtt{SES}^* = (\mathtt{LLSet}^*, \mathtt{LL}^*_c, \mathtt{FM}^{'*})$ as follows:

- Game_0: This game is equivalent with the $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ game.
- Game_1: This game behaves as the $\mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ game with the difference that $\mathcal{S}$ does not have access to S. The simulator through the $\mathcal{L}_1$ leakage function builds the substring encrypted structure $\mathtt{SES}$ as follows: We assume the existence of an algorithm $S \leftarrow \mathsf{Build}(n', str)$, which takes as input $n' \in \mathbb{N}$ and the structure $str = \{c\}_{i=1}^{n'}, c \in \Sigma^*$ and outputs a bitstring of length n', from a vocabulary Σ^*. Notice that as in the real game the valid length of the original stream is nor revealed and only the length of the string after the padding n' is leaked. $\mathcal{S}$ selects uniformly at random keys $\mathsf{k} = (\mathsf{k_f}, r, \mathsf{k}_\pi, \mathsf{k_e})$ for a PRF $\mathsf{F}_{\mathsf{k_f}} : \{0,1\}^\lambda \times \{0,1\}^\nu \to \{0,1\}^\mu$, a PRP $\Pi_{\mathsf{k}_\pi} : \{0,1\}^\lambda \times \{0,1\}^\nu \to \{0,1\}^\nu$ and a symmetric encryption algorithm $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ and runs $\mathtt{SES} \leftarrow \mathsf{PreProcess}(\mathsf{k}, \mathsf{Build}(\mathcal{L}_1))$. $\mathcal{S}$ uses $\mathsf{F}_{\mathsf{k_f}}$ to evaluate bit strings of length c_n $\mathcal{L}_2(q) = c_n, str$.
- Game_2: This game behaves similarly with Game_1, but we replace the $\mathsf{F}_{\mathsf{k_f}}$ with a real random function which is evaluated through access to an oracle $\mathcal{O}^{\mathsf{RF}}(\lambda, \mu, \nu)$.
- Game_3: This game behaves similarly with Game_2, but we replace the Π_{k_π} with a real random permutation which is evaluated through access to an oracle $\mathcal{O}^{\mathsf{R}\Pi}(\lambda, \nu)$.
- Game_4: In Game_4 we replace the semantically secure $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ with real random values by querying an oracle $\mathcal{O}^{\mathsf{RE}}(\lambda)$.

We write $\mathsf{Game}_i \approx \mathsf{Game}_j$ to denote that the view of probabilistic polynomial time adversary $\mathcal{A}$ is indistinguishable between the output of Game_i and Game_j. $\mathsf{Game}_0 = \mathbf{Real}^{\mathsf{S^3E}}_{\mathcal{A}(\lambda)}$ by definition, $\mathsf{Game}_1 \approx \mathsf{Game}_0$ as long as no collisions happen to the evaluation of $\mathsf{F}_{\mathsf{k_f}}$, Π_{k_π}, $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ or E, $\mathsf{Game}_2 \approx \mathsf{Game}_1$ as long as $\mathsf{F}_{\mathsf{k_f}}$ is indistinguishable from real random function, $\mathsf{Game}_3 \approx \mathsf{Game}_2$ thanks to the indistinguishable output of Π_{k_π} from real random permutations, $\mathsf{Game}_4 \approx \mathsf{Game}_3$ because of the semantically secure $\mathsf{SKE} = \{\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\}$ and finally $\mathsf{Game}_5 = \mathbf{Ideal}^{\mathsf{S^3E}}_{\mathcal{A},\mathcal{S}(\lambda)}$ by definition.