Oblivious Substring Search with Updates

Tarik Moataz[1] and Erik-Oliver Blass[2]

[1]Telecom Bretagne, IMT, France,
and Colorado State University, CO, USA,
tmoataz@cs.colostate.edu
[2]Airbus Group Innovations, 81663 Munich, Germany,
erik-oliver.blass@airbus.com

Abstract

We are the first to address the problem of efficient oblivious substring search over encrypted data supporting updates. Our two new protocols SA-ORAM and ST-ORAM obliviously search for substrings in an outsourced set of n encrypted strings. Both protocols are efficient, requiring communication complexity that is only poly-logarithmic in n. Compared to a straightforward solution for substring search using recent "oblivious data structures" [30], we demonstrate that our tailored solutions improve communication complexity by a factor of $\log n$. The idea behind SA-ORAM and ST-ORAM is to employ a new, hierarchical ORAM tree structure that takes advantage of data dependency and optimizes the size of ORAM blocks and tree height. Based on oblivious suffix arrays, SA-ORAM targets efficiency, yet does not allow updates to the outsourced set of strings. ST-ORAM, based on oblivious suffix trees, allows updates at the additional communications cost of a factor of $\log\log n$. We implement and benchmark SA-ORAM to show its feasibility for practical deployments: even for huge datasets of 2^{40} strings, an oblivious substring search can be performed with only hundreds of KBytes communication cost.

1 Introduction

Users outsource their data to clouds to benefit from reduced costs and availability. However, several concerns related to confidentiality create considerable uncertainty for those maintaining sensitive information. There are several devastating attacks on clouds known that have led to data disclosure and sensitive information theft [17, 28]. A standard solution is to store only encrypted data in the cloud. While this approach preserves users' confidentiality, it impedes typical cloud applications, such as searching on outsourced data.

Recent research has investigated the topic of efficiently searching over encrypted data. In particular, single exact keyword search schemes, e.g., [9, 26] and many others, have been presented. These techniques fall within symmetric searchable encryption (SSE) that enables efficient sublinear search complexity, but at the cost of "leaking" some information about the search, such as the search and access *pattern*.

To strengthen security, one can adopt more powerful cryptographic primitives that prevent from leaking access patterns, such as oblivious RAM (ORAM) [23] or Private Information Retrieval (PIR) [24]. In particular, the former technique has been thoroughly investigated during recent years leading to many schemes with poly-log communication complexity [27].

In this paper, we are the first to investigate oblivious substring search over encrypted data with security properties similar to those of ORAM. Our goal is to not leak any information about the search pattern. More precisely, given a set of n strings $\mathcal{S} = \{s_1,...,s_n\}$ and a substring ς, the search returns the number of occurrences of substring ς in the set of strings $\mathcal{S}$. We will provide *obliviousness*, ensuring that the server performing the verification will (1) neither learn which substring has been queried for, (2) nor learn which string matches the substring, (3) nor learn the outcome of the search; only the user understands the result. Our second goal is to handle dynamic string *updates*, i.e., to allow the set $\mathcal{S}$ to increase.

Motivation Enabling efficient substring search over encrypted data is important in many real-world applications. For example, imagine a company's web proxy logging

millions of URLs accessed by company workstations every day. Securely storing a large amount of accessed URL log data should not impede possible future forensic analysis to detect system compromise. A typical forensic analysis could include, e.g., searching for suspicious URLs or parts (substrings) of URLs accessed. Trivial solutions to this problem like generating all substrings of possible keywords (URLs) in the logs and storing them using SSE techniques or ORAM result in significant overhead. The drawbacks of trivial solutions are twofold: (1) the considerable amount of storage induced, and (2) the large communication complexity. Recently, Chase and Shen [8] introduced a first, efficient solution to substring search based on suffix trees. For security parameter λ, ς being the substring searched for, and occ the number of occurrences of the substring, their scheme offers low search complexity over encrypted data in $O(\lambda \cdot \varsigma + \mathsf{occ})$.

While highly efficient, Chase and Shen [8] leak sensitive information to an adversary: the pattern of the prefix of the substring and the occurrences of positions induced by the leaf and index intersections. Therewith, an adversary (server) will know whether two substring searches are sharing the same prefix. This leakage is a result of the underlying suffix tree structure which inherently discloses information of its structure. A server with some knowledge of the dictionary and a certain number of queries can recover the content of many edges of the encrypted suffix tree only based on the frequency of letters distribution. While this leakage by Chase and Shen [8] may be considered acceptable in some scenarios, this paper targets stronger security protection.

Contribution In this paper, we present the first oblivious substring search over encrypted data. Roughly speaking, we construct a scheme that efficiently inserts a pre-processed suffix data structure into a modified, tree-based ORAM. While this can also trivially be achieved using very general solutions [30], we show that our new, dedicated oblivious data structure improves by a factor of $\log n$ in communication over the general solution. Therewith, even for a huge number of strings, e.g., $n = 2^{40}$, testing obliviously for a matching substring requires communication in the order of only 100 KByte.

First, we present a static construction, dubbed SA-ORAM, that enables oblivious substring search for an a priori fixed set of strings. SA-ORAM is based on incorporating a *suffix array* construction on the position map of a tree-based ORAM. The position map is a recursive structure composed of a number of trees with linearly-increasing heights. Searching for a substring on plaintext data using suffix arrays is based on binary search. For a number of strings n, string length $m \in \Omega(\log n)$, and size of the alphabet $\gamma \in \Omega(\log n)$, searching for a substring using SA-ORAM can be performed with communication complexity $O(\log^3 n) \cdot \omega(1)$.

Static constructions only work for a fixed set $\mathcal{S}$ of strings. Consequently, our second contribution, dubbed ST-ORAM, enables to obliviously add strings to $\mathcal{S}$. To search for a substring, communication complexity is in $O(\log^3 n \cdot \log\log n) \cdot \omega(1)$. ST-ORAM incorporates suffix trees in a recursive way. The suffix tree data dependency implies that the distribution of leaves over the levels can vary depending on the data set. To tackle this issue, we present a new *suffix tree encoding* fixing the distribution of leaves and interior nodes in a suffix tree.

Table 1 provides a comparison between general purpose oblivious data structure (ODS) [30] and our dedicated constructions, SA-ORAM and ST-ORAM. The table shows the asymptotic gain under different settings of the size of the alphabet γ and length of the string m.

2 Background

In this section, we briefly revisit tree-based ORAMs. In particular, we will focus on Path ORAM [27].

First, the objective of oblivious RAM in general is to hide access patterns. That is, ORAM insures that, for an adversary, any two accesses to the same or different data will look indistinguishable. The data is divided in a number of blocks (elements) that we denote by n. Each block of data has a specific size in bits. Our focus will be on tree-based ORAMs that are composed of two main parts: the position map and the main data ORAM. The position map consists of a logarithmic number of tree ORAMs that map an ORAM address a to a tag. This tag identifies a leaf in the main data ORAM. Any node in the tree is called a bucket, where each bucket is composed of a constant number of entries z. The difference between a tree in the position map and the main data ORAM is the size of blocks and the tree height. In the position map, the block size is fixed to $O(\log n)$, while the block size in the main data ORAM can vary.

Tree-based ORAMs simulate $\mathsf{Read(a)}$, $\mathsf{Write(a,data)}$ by the following three operations: $\mathsf{ReadAndRemove(a)}$, $\mathsf{Add(a,data)}$, and Evict. Parameter a is the address of the element and data is the data to be written. Any $\mathsf{ReadAndRemove}$ operation is followed by an Add operation. In Path ORAM, the eviction process is performed at the same time the write operation is performed. We

Scheme	$m,\gamma \in \Omega(1)$	$m,\gamma \in \Omega(\log n)$	$m,\gamma \in \Omega(\log^2 n)$
SA-ODS	$O(\log^3 n)\cdot\omega(1)$	$O(\log\log n\cdot\log^3 n)\cdot\omega(1)$	$O(\log\log n\cdot\log^4 n)\cdot\omega(1)$
ST-ODS	$O(\log^2 n)\cdot\omega(1)$	$O(\log^4 n)\cdot\omega(1)$	$O(\log^6 n)\cdot\omega(1)$
SA-ORAM	$O(\log^3 n)\cdot\omega(1)$	$O(\log^3 n)\cdot\omega(1)$	$O(\log\log n\cdot\log^3 n)\cdot\omega(1)$
ST-ORAM	$O(\log^2 n)\cdot\omega(1)$	$O(\log\log n\cdot\log^3 n)\cdot\omega(1)$	$O(\log\log n\cdot\log^4 n)\cdot\omega(1)$

Table 1: Communication complexity of oblivious substring schemes.

overview these operations.

ReadAndRemove(a): To access element with address a, the user first fetches the $tag\ t$ of the element from the position map. The user then downloads the entire path that starts from the root to the leaf tagged with t. The user decrypts all buckets, retrieving the desired block which has a as its identifier. Finally, the user appends the downloaded path to a *stash* of size $O(\log n)$. The stash is stored on the user side.

Add(a,data): The data associated to address a will be overwritten by data. From the stash, the user recursively determines elements that are closest to the leaf tagged with t and outputs a sorted path which is sorted based on the least common ancestor rule. This Evict operation needs a logarithmic number of loops throughout the stash to compute the path to be inserted. The same tag t retrieved during a ReadAndRemove operation will be used for the path insertion. All elements are IND-CPA encrypted.

In this paper, we will be using the position map of Path ORAM as a building block for both our static and dynamic solution.

2.1 OSS definition

Oblivious substring search (OSS) is composed of three probabilistic algorithms:

- $(\mathsf{OSS},\kappa)=\mathsf{SetupOSS}(\mathcal{S},n,m,\gamma,\lambda)$: takes as an input the set of strings $\mathcal{S}$, the maximum number of strings n, the maximum string length m, the alphabet size γ, and security parameter λ. It initializes a new instance OSS and outputs secret state κ and OSS.
- $\mathsf{OSS}=\mathsf{AddString}(s,\kappa)$: takes as an input a new string s such that $|s|\leq m$, secret state κ, and adds s to OSS. It outputs the updated OSS.
- $\mathsf{occ}=\mathsf{QuerySubstring}(\varsigma,\kappa)$: takes as an input the substring ς and secret κ. It outputs the occurrences $\mathsf{occ}\in\mathbb{N}\cup\{\perp\}$

We say that OSS is correct if for all λ, m, n and $\gamma \in \mathbb{N}$, for all $\mathcal{S} \in [\gamma]^{m\times n}$, for all (OSS,κ) output by $\mathsf{SetupOSS}(\mathcal{S},n,m,\gamma,\lambda)$, for all substrings $\varsigma \in [\gamma]^{\leq m}$, $\mathsf{QuerySubstring}(\varsigma,\kappa)$ outputs the correct number of occ with all but negligible probability.

2.2 Security definition

Oblivious substring search (OSS) should preserve the following obliviousness requirement.

Definition 2.1. *Let $\overrightarrow{a} = \{(op_1,d_1),(op_2,d_2),\ldots,(op_M,d_M)\}$ be a sequence of M accesses (op_i,d_i), where op_i denotes* AddString *or* QuerySubstring*, and d_i denotes the data to be written. Data d_i equals string s_i if $op_i=$ AddString*, and substring ς_i otherwise. Let $A(\overrightarrow{a})$ be the access pattern induced by sequence $\overrightarrow{a}$, λ the security parameter, and $\epsilon(\lambda)$ is a negligible function in λ. We say that* (SetupOSS, AddString, QuerySubstring) *is secure* iff*, for any PPT adversary $\mathcal{D}$ and any two same-length sequences $\overrightarrow{a}$ and $\overrightarrow{b}$, access patterns $A(\overrightarrow{a})$ and $A(\overrightarrow{b})$,*

$$|Pr[\mathcal{D}(A(\overrightarrow{a}))=1]-Pr[\mathcal{D}(A(\overrightarrow{b}))=1]|\leq\epsilon(\lambda)$$

holds.

3 SA-ORAM: Suffix arrays over ORAM

3.1 Suffix Arrays

Suffix arrays [22] are sorted index arrays that enable searching for substring occurrences in a given string with search complexity $O(\varsigma+\log m)$. While the search complexity increases by an additional $\log$ factor over suffix trees, their storage efficiency is their advantage [31]. Moreover, their construction is very elegant and simple. We now give an example of a suffix array construction. Consider the string security.

1. First, we extract all suffixes from right to left, and we enumerate them: (1 - y), (2 - ty), (3 - ity), (4 - rity), (5 - urity), (6 - curity), (7 - ecurity), and (8 - security).

2. The second step consists of sorting these suffixes in lexicographic order: (6 - curity), (7 - ecurity), (3 - ity), (4 - rity), (8 -security), (2, ty), (5 - urity) and (1 - y). Based on the lexicographically order, this array is transformed into a (balanced) binary search tree BST, where each node stores its index, see Fig 1.

3. Finally, we delete all the suffixes, keep the positions in the array (6,7,3,4,8,2,5,1), and store the string security.

Storing the sorted array without the BST is sufficient to emulate a BST – we can perform a simple binary search over that array. For ease of exposition, we will stick in our description to BST, though. To search for the substring ς, we retrieve the index i stored in the root and compare the i^{th} letter of security to the first letter of ς. If the letters match, we move to the $(i{+}1)^{\text{th}}$ letter and check it against the second letter of the substring ς. Otherwise, we move to left or right child of the root, and we redo the same process. Thus, the search for any substring can be performed in $O(\varsigma \cdot \log m)$ steps. This complexity can be improved by adding the longest common prefix information, so a search can be performed in $O(\varsigma{+}\log m)$, see Manber and Myers [22] for details.

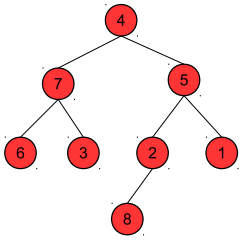

Figure 1: Illustration of suffix array BST of security.

3.2 SA-ORAM overview

The above lexicographically sorted array is equivalent to constructing a binary search tree BST. Obliviously searching in a BST has been recently introduced by Gentry et al. [14]. Therewith, Wang et al. [30] create oblivious AVL trees. The search in a perfectly balanced BST is logarithmic in the number of stored elements, as each level of the BST can be stored in a level of the position map. Thus, obliviously accessing an element in the position map will add a multiplicative factor of $\log n$ due to the height of the tree. Recall that each level of the position map is a tree with a height at most equal to $\log n$.

Tree-based ORAMs are one of the most efficient ORAM structures with poly-logarithmic worst-case communication complexity. Path ORAM allows for a search complexity in $O(\log n)$, having blocks of size $\Omega(\log^2 n)$ and user-memory in $\Theta(\log n)$. The main difference between our setting and Path ORAM is that we do not want to *store* data, but only the BST that will spread out throughout the entire level of the position map. Employing the idea by Gentry et al. [14], including a BST on the top of the position map will not increase communication complexity, and we can maintain exactly the same asymptotics as using the position map alone.

We now briefly compute the exact communication complexity induced by the position map. The position map is a recursion of ORAMs that linearly increase their height. The position map maps n addresses to n tags. It is composed of a logarithmic number of tree ORAMs, ORAM_i, for $i \in [\frac{\log n}{\log \tau}]$. ORAM_{i+1} has τ times more leaves than ORAM_i. τ is called the recursion factor, and $n{=}\tau^l$. Thus, the height of ORAM_i equals $\log \tau^i$. Parameter z denotes the number of entries in each bucket of ORAM_i. The communication complexity to retrieve the tag computes to:

$$\begin{aligned}\sum_{i=2}^{l-1} z{\cdot}\tau{\cdot}\log\tau^i{\cdot}\log\tau^{i+1} &= z{\cdot}\tau{\cdot}\log^2\tau \sum_{i=2}^{l-1} i{\cdot}(i{+}1) \\ &= z{\cdot}\tau{\cdot}\log^2\tau{\cdot}\left(\frac{l(l{-}1)(l{+}1)}{3} - 2\right)\end{aligned}$$

This complexity sums up a logarithmic number of paths with different heights. The equation can be rewritten to be in function of one variable, for example by replacing l by $\frac{\log n}{\log \tau}$. We can optimize the communication complexity by finding the best value of l that minimizes the above equation for any $l \geq 3$. For $\tau = 16$, $n = 2^{32}$, $l = 8$, and $z = 5$. The position map then exactly costs 1660 Bytes which is independent of the size of the element (block), but depends only on the number of stored elements. Also, the communication complexity increases poly-logarithmically in the number of elements. This remark is very important because it shows us that we can build a BST with a practically small amount of bits.

We have shown that a position map alone does not consume a lot of bandwidth since it is independent of the data block size. SA-ORAM will make use of the position map as a building block to store the nodes of the suffix array BST. Roughly speaking, each level of the BST will be stored in the associated tree of the position map. However, this will not be straightforward and some modifications on the BST are needed beforehand.

For plaintext substring search over suffix arrays, we use the BST and the stored string(s) to verify whether there is

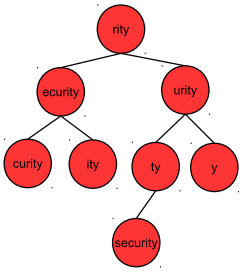

Figure 2: Illustration of augmented BST of the word security.

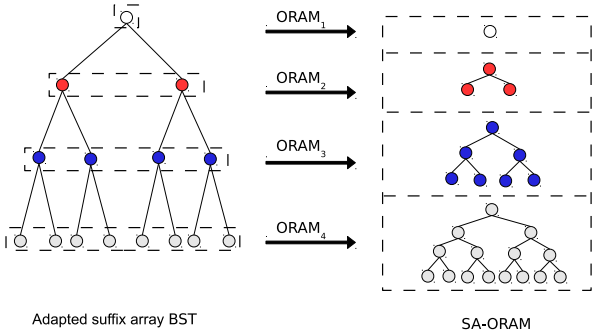

Figure 3: High level illustration of SA-ORAM from a suffix array BST with $\tau = 2$.

a match or not. First, for each level of the BST, we retrieve the index from the corresponding node. Second, we verify if the substring exists by checking the stored string(s) starting from the position equal to the index retrieved in the first step. These two operations are performed a logarithmic number of times. For oblivious substring search, we cannot disassociate these two steps, otherwise the server can always distinguish which string has been accessed. A trivial solution consists of storing the string in a data ORAM, e.g., Path ORAM. However, with a large number of strings, this will require considerable bandwidth and makes the scheme very costly. To solve this issue, we create a suffix array BST that also contains the suffixes instead of only the position within each node of the BST. The only difference in our case consists of avoiding the last step in suffix arrays construction (step 3) that eliminates the suffix information, i.e., our construction is only based on step 1 and 2. Finally, we generate a BST where each node can be a suffix. See Fig 2 for this "augmented" tree of security.

To sum up, SA-ORAM is a recursive set of trees where each tree is an ORAM, see Fig 3. Each level of the BST, containing l nodes, is represented as an ORAM with l leaves. The number of trees is logarithmic in the number of suffixes. The logarithmic number of trees equals the height of the suffix tree BST. The construction consists of storing the nodes of each level in the corresponding level of the recursive SA-ORAM. To search for a substring, the user has to access a logarithmic number of trees that will lead to a match. In SA-ORAM, a match can occur (say) in the 2^{nd} level, but the user has to continue until the last tree to preserve obliviousness.

SA-ORAM handles occurrences search, i.e., retrieving the number of suffixes that contain the substring searched for. Similarly to plaintext search over suffix arrays, SA-ORAM enables oblivious occurrences search based on two binary searches on the SA-ORAM, we can bound the existence of all substrings to a given interval. Since the suffix array is lexicographically ordered, keeping meta-information about the order of the suffix is enough to find the number of occurrences. Particularly, the first binary search determines the smallest suffix that has the substring while the second determines the largest. Getting these two positions, we can determine all other substrings that match the substring since they will be within this interval. Retrieving the occurrence information is very important for any user who wants to use our scheme as a component to search over encrypted data. We now detail the algorithm SetupOSS that outputs SA-ORAM main structure.

3.3 SetupOSS: SA-ORAM setup phase

Let $\mathcal{S} = \{s_1, \cdots, s_n\}$ be the list of n strings such that the maximal string length equals m. Σ is the alphabet set that has size γ. First, we generate all suffixes suf_i associated to $\mathcal{S}$. Let τ be the recursion factor and N the total number of generated suffixes such that $N = \tau^l$. The number of unique strings n gives an upper bound for the suffix numbers N such that $N \leq n \cdot m$. Let $\mathsf{O} = \{(\mathsf{suf}_1, 1), \cdots, (\mathsf{suf}_N, N)\}$ denote the suffix set where each suffix is associated to its position. The set O is lexicographically sorted and therefore can be represented in a τ-ary search tree .

Overview: Having a sorted list is equivalent to a BST and more generally to τ-ary search tree. Constructing such a tree represents a $l = \frac{\log N}{\log \tau}$ τ-chotomy of the entire set O. For example, if $\tau = 2$, the set O will be first divided in two. Then, we insert only the two minimum values of both subsets in two nodes that will represent the first level of the tree. We divide a second time the entire set O and we insert now the four minimum values in the four leaves. We reiterate the process a logarithmic number of time. As illustrated in Fig 3, we will first generate a τ-ary search tree and gradually insert the τ^i nodes of a level i of the BST in

an ORAM with τ^i elements. The elements of the set O will be evenly distributed over the l trees of the SA-ORAM.

Details: More precisely, the construction phase is composed of l steps. For the first step, we divide the set O in τ parts such that for

$$o_i = \{(\mathsf{suf}_{\frac{(i-1)\cdot N}{\tau}+1}, \frac{(i-1)\cdot N}{\tau}+1), \cdots, (\mathsf{suf}_{\frac{i\cdot N}{\tau}}, \frac{i\cdot N}{\tau})\},$$

we have

$$\mathsf{O} = \{o_1, \cdots, o_\tau\}.$$

We continue the process for the second level where we divide each o_i in τ parts, compute the minimum for each new subset and redo the same process until the l^{th}step.

To sum up, $o_{i,j}$ represents the i^{th} set in the j^{th} step for $i \in [\tau^j]$ and $j \in [l-2]$ such that:

$$o_{i,j} = \{(\mathsf{suf}_{\frac{(i-1)\cdot N}{\tau^j}+1}, \frac{(i-1)\cdot N}{\tau^j}+1), \cdots, (\mathsf{suf}_{\frac{i\cdot N}{\tau^j}}, \frac{i\cdot N}{\tau^j})\}.$$

We define $\min o_{i,j}$ as the smallest suffix in the set $o_{i,j}$ and since $o_{i,j}$ is an ordered set, then $\min o_{i,j} = \mathsf{suf}_{\frac{(i-1)\cdot N}{\tau^j}+1}$. We define $\mathsf{O_j}$ to be the set that contains the minimum values for the j^{th} step and the position of the set in the next level of the position map. O_j represent the sets that will be stored in the position map such that for each $j \in [l-1]$, $r_{i,j} \xleftarrow{\$} [\tau^{j+1}]$, and $\min o_{i,j} = \mathsf{suf}_{\frac{(i-1)\cdot N}{\tau^j}+1}$

$$\begin{aligned} \mathsf{O}_j &= \{\mathsf{O}_{i,j}\}_{i\in[\tau^{j-1}]} \\ &= \{(\min o_{i,j}, r_{i,j}), \cdots, (\min o_{i+\tau-1,j}, r_{i+\tau-1,j})\}_{i\in[\tau^{j-1}]} \end{aligned}$$

For $j = l$, we have $\mathsf{O}_l = \{o_{i,l}\}_{i\in[\tau^{l-1}]}$.

The user has to insert the sets O_j for $j \in [l]$ in the position map. The position map, as previously introduced, is divided in l ORAMs such that ORAM_j is a binary tree that has τ^{j-1} leaves. ORAM_1 consists of one block that will be kept locally stored on the user side. We assume that all these ORAMs are instantiated empty and contain dummy elements. To store O_j in ORAM_j, we proceed as follows for $1 < j \leq l$

- For each subset $\mathsf{O}_{i,j-1} \in \mathsf{O}_{j-1}$ for $i \in [\tau^{j-2}]$
  - For each $v_t = (\min_t, tag_t) \in \mathsf{O}_{i,j-1}$ for $t \in |\mathsf{O}_{i,j-1}|$, $\mathsf{Write}(\mathsf{O}_{i+t,j}, tag_t)$ in ORAM_j

Write is an ORAM operation that simulates a ReadAndRemove followed by and Add. Finally, the user outsources the encoded position map, SA-ORAM= $\{\text{ORAM}_i\}_{1<i\leq l}$ to the server. Note that $\text{ORAM}_1 = \mathsf{O}_1$ is stored on the user side.

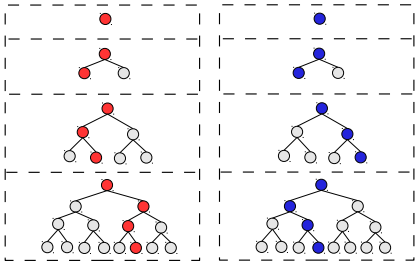

Figure 4: High level illustration of SA-ORAM substring query where all red and blue buckets are downloaded and uploaded.

3.4 QuerySubstring: SA-ORAM search phase

We want to search for the substring ς and find all occurrences, i.e., all strings that contain the substring. In plaintext suffix arrays, this search can be performed by two consecutive binary searches, finding therefore the range of ς.

From a high level perspective, to search for a substring ς, the user recursively accesses all ORAM_i for $i \in [l]$. Each access outputs a pointer to the next ORAM in such a way the user is obliviously performing a binary search. In ORAM_1, the user first checks out the suffix interval that eventually contains the searched for substring. Once the user determines the interval, he retrieves the pointer which is a path identifier of the next ORAM tree. The user repeats in logarithmic time these steps until the last tree ORAM_l. Finding out the number of occurrences of a substring can be done similarly to the plaintext search. The user will access SA-ORAM twice, the first access will output the smallest suffix that contains the substring while the second access outputs the largest suffix that contains the substring, see Fig 4. The order relation used here is the lexicographic order. Finally, the user outputs the number of occurrences which equals the difference of the retrieved block indexes.

We define two conditions that capture the binary search over the position map. For θ and θ_t in $\{\min, \{\mathsf{suf_i}\}_{i\in[N]}\}$: "$\mathsf{cond}_1(\theta) = \theta_t \leq \varsigma < \theta_{t+1}$" and "$\mathsf{cond}_2(\theta) = \theta_t < \varsigma \leq \theta_{t+1}$". The user performs the following steps for each $i \in \{1,2\}$ (this represents two binary searches over SA-ORAM):

- Initialize two variables temp and tag. Locally, the user compares in alphabetical order ς and each element $(\min_t, r_t) \in \mathsf{O}_1$ for $t \in [\tau]$ such that if $\mathsf{cond_i}(\min_t) = \mathsf{true}$ then $\mathsf{temp} \leftarrow \min_t$, $\mathsf{tag} \leftarrow r_t$ and $r_t \xleftarrow{\$} [\tau]$

- For each $1<j\leq l-1$ do:
  - Read(tag) from ORAM$_j$ and retrieve the block B that has temp as its tag. Compare each element $(\min_t, r_t) \in$ B for $t \in [\tau]$ such that if cond$_i(\min_t)$=true then temp $\leftarrow \min_t$, tag$\leftarrow r_t$ and $r_t \xleftarrow{\$} [\tau]$
  - Write(B,tag)
- For $j = l$, Read(tag) from ORAM$_l$ and retrieve the block B that has temp as its tag. Compare each element $(\text{suf}_t, \text{num}_t) \in$ B for $t \in [\tau]$ such that if cond$_i$(suf$_t$)=true then temp $\leftarrow$ numb$_t$ and Write(B,tag).

In a match case, the user knows the number of occurrences based on the indexes stored in the SA-ORAM blocs. In fact, the block retrieved in the first search contains the suffix index which is the suffix rank in the ordered list O while the second one contains the rank of the second suffix. Therefore, the user respectively retrieves two indexes a_1 and a_2 from the first and second search. The number of occurrences occ is defined as $a_2 - a_1$.

3.5 Block compression

The search complexity of SA-ORAM improves upon the straightforward approach by Wang et al. [30] by only multiplicative constant but the asymptotics remain the same, namely $O(m \cdot \log\gamma \cdot \log^2 N)$. Fortunately, SA-ORAM's hierarchical structure enables us to decrease the amount of information that we are storing in each block. Using suffix arrays over ORAM, we can improve by a $\log N$ factor by taking advantage of the data dependency and suffix ordering, for any $m,\gamma \in \Omega(\log n)$.

Having a hierarchical structure, we can avoid storing the entire suffix in each level which will decrease the size of each block by $m\log\gamma$. One might assume that storing only one character in all recursive ORAMs, except the last one, is sufficient to lead the binary search, however it is not totally correct. For $\tau \geq 2$, the pointers within an interval in a given block can share a common prefix (for example suf$_i$ and suf$_{i+1}$ share the same prefix) and therefore one character will be not enough. While we want to avoid storing unnecessary information, having a counter of the length of common substring framed by two characters is sufficient for the search. Keep in mind that the user will verify whether the substring he is searching for exists or not in the last level of the recursion. This simple modification reduces the suffix size within each block, except blocks in the last ORAM, from $m\log\gamma$ to $\log m \cdot \log\gamma$, see Fig 5.

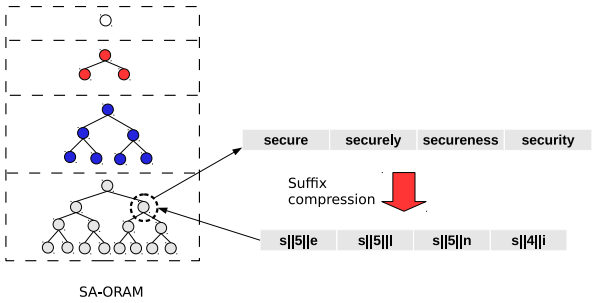

Figure 5: Example of suffix compression

4 ST-ORAM: suffix trees over ORAM

Motivation: Our static SA-ORAM only supports a static set of suffixes. Now, we focus on a dynamic construction that will enable efficient adding of new strings. Our dynamic construction is based on the same philosophy used in SA-ORAM, but with a different representation of the suffix data structure. Updating suffix arrays in a binary tree representation can be easy, if we use self-balancing trees such as AVL or Red-Black trees. However, our representation associates each node to a specific recursion level. Therefore, if we perform a rotation to balance the tree after every insertion, many nodes in the tree change their levels resulting an inefficient construction.

To mitigate this problem, related work stores the suffix array BST in an oblivious data structure (ODS) [30]. This allows updates to the structure, as ODS can handle self-balancing. However, using an ODS would imply higher communication complexity.

In our approach, instead of using suffix arrays, we store a suffix tree in an ORAM. Similarly to SA-ORAM, we associate each level of the suffix tree to a level of ORAM recursion. The advantage of a suffix tree over suffix arrays is its constant height. Suffix trees have a height equal to the longest string m, while in a suffix array, the height depends on the number of elements. Therefore, a node created in a suffix tree will always keep the same position, i.e., the same level in the tree. However eventually, the node's value can change through updates. Again, one could store a suffix tree in the ODS structure, however it will be at the expense of storing the maximum number of leaves of a suffix tree as well as very large blocks which will be inefficient. The total number of leaves of the ODS will equal the total number of nodes in an entire suffix tree which is upper bounded by $2n \cdot m$. Also, the block size of a suffix tree stored over ODS

will be large, also increasing communication overhead.

In our new construction ST-ORAM, we hide information about updates from the server. For this, we define the maximum size of strings during setup time, i.e., each tree is instantiated with a maximum number of leaves also considering future updates. Thus, even if an element is added, the tree will keep the same size. For SA-ORAM, even if we fix the number of leaves of each recursive tree to be equal to an upper bound, there is no theoretical guarantee that the number of elements will not exceed this bound. The maximum size can be considered an upper bound of the number of strings that the system can handle for a defined period of time. For example, to handle, e.g., 10^6 strings per day for 10 years, we fix the maximum number of strings to $n \approx 10^9$ (assuming all strings are different).

This approach suffers from some trivial drawbacks related to storage efficiency and scalability. Also, in some cases, it is difficult to define the proper upper bound of strings that will be stored in the structure. To mitigate, we can relax update security by disclosing new memory allocation. That is, all trees will contain only the number of nodes required for the current number of strings stored. Any update involving the insertion of a new string will imply the creation of new nodes in the corresponding trees, increasing its size. Simply by looking at the size, the server can therefore distinguish between a normal add/read and an update. Resizing the construction will achieve optimal communication complexity. For each level, we will have the exact number of nodes required. In this paper, we only consider the case where the ORAM has an upper bound for each recursive tree. We target a scheme that enables updating the oblivious structure while preserving the obliviousness against any query type.

4.1 Suffix trees

A suffix tree [31] is a trie-like representation of text supporting a wide range of applications on strings. In particular, it is suited for substring (pattern) search. Similar to suffix arrays, suffix trees are pre-processed data structures that enable sub-linear search in the size of the stored strings. Suffix trees contain all possible suffixes of a given string. This allows search complexity for a substring ς to be in $O(\varsigma)$. The time complexity to construct a suffix tree is in $O(m)$ using, e.g., Ukkonen's on-line construction [29]. As Ukkonen's algorithm is very involving and out of scope of this paper, we briefly discuss only a straightforward construction of suffix trees with time complexity in $O(m^2)$.

Construction: We consider a set of strings $S = \{s_1, \cdots, s_n\}$ such that $\max_{i \in [n]} |s_i| = m$. First, we generate for each string all possible suffixes. Let $\mathsf{Suf}(s_i)$ denote the set of suffixes associated to s_i. All suffixes in $\mathsf{Suf}(s_i)$ are unique. We define two operators cont and diff that respectively output the common prefix between two strings and the inclusive difference between two suffixes. For instance, considering the following three strings $s_1 = \mathsf{aabca}$, $s_2 = \mathsf{aabac}$ and $s_3 = \mathsf{aab}$, the operators output $\mathsf{cont}(\mathsf{s_1},\mathsf{s_2}) = \mathsf{aab}$ and $\mathsf{diff}(\mathsf{s_2},\mathsf{s_3}) = \mathsf{ac}$. The diff operator is only defined when the first input is a prefix of the second one. To store a string s_i in the suffix tree, for each suffix $\mathsf{suf_i} \in \mathsf{Suf}(s_i)$, we perform:

- Search for $\mathsf{suf_i}$ in the suffix tree and retrieve the path s that has an overlap with $\mathsf{suf_i}$. The path s is a previously stored suffix. The path s is composed of at least one edge and at most m edges. An edge is a set of characters belonging to the alphabet Σ
- Compute $\mathsf{cont}(\mathsf{suf_i},\mathsf{s}) = \mathsf{pref}$. If $|\mathsf{pref}| > 0$, create a vertex at the end of the prefix pref that will have as a first edge $\mathsf{diff}(\mathsf{pref},s)$ and a second edge which is equal to $\mathsf{diff}(\mathsf{pref},\mathsf{suf_i})$. Otherwise, create one edge that has $\mathsf{suf_i}$ as its value starting from the root, see Fig6
- Each path ends with a vertex that contains a linked list of the suffix identifiers as well as their positions in each string

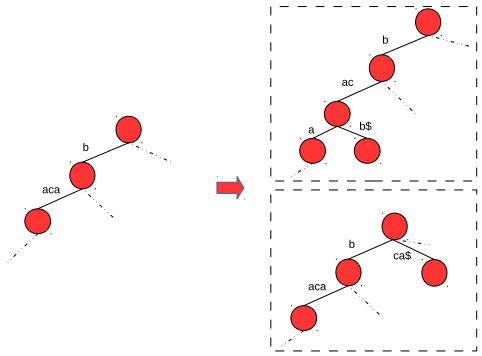

Figure 6: Illustration of two cases of suffix insertion. The first one depicts the insertion of the suffix bacb. In this case, $\mathsf{pref} = \mathsf{bac}$, $\mathsf{diff}(\mathsf{pref},s) = \mathsf{a}$ and $\mathsf{diff}(\mathsf{pref},\mathsf{suf_i}) = \mathsf{b\$}$. The second case represents the insertion of ca\$ for which $\mathsf{pref} = \emptyset$.

To later add new strings, we perform the same steps above for the new suffix. The resulting suffix trees display the following properties.

- Each internal node has at least two and at most γ children

- The total number of nodes is at most $2 \cdot N$ for N suffixes
- Each path starting from the root towards a leaf represents one unique suffix
- For N suffixes, the suffix tree has exactly N leaves

It is important to concatenate a special character $\$ \notin \Sigma$ at the end of each string. That will avoid that a suffix is a prefix of another suffix. For example, string aacaba has a as prefix of substring aacaba. See Fig 7 for an example of a suffix tree construction.

To search for all occurrences occ of a substring ς, we must first find the path that contains ς. Two cases might arise. The first one is that the substring ς exactly ends on an interior/leaf node that we denote as our *reference*. The reference defines a root of a subtree. The leaves of this subtree are the suffixes that contain the substring ς. The second case occurs when ς ends in the middle of an edge in the suffix tree. In this case, we consider the lower node linked by this edge as our nodes' reference. In both cases, the number of occurrences equals the number of leaves in the subtree rooted by the node's reference. We can order the leaves of the suffix tree lexicographically as shown in Fig 7, so we can point to an ordered set of leaves only by two pointers. Therefore, for each interior node, we store two pointers to refer to all the leaves of the subtree rooted by this node. Thus, to search for the occurrence of a substring ς occurring occ times in the suffix tree, we need $O(\varsigma+\mathsf{occ})$ node's access.

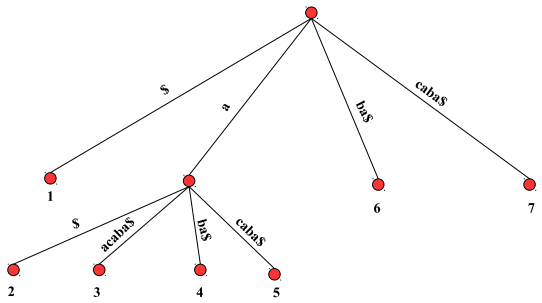

Figure 7: Illustration of suffix tree construction using the string aacaba.

4.2 ST-ORAM Overview

We now present our solution to efficiently build a suffix tree over ORAM. Before describing the intuition behind this second protocol called ST-ORAM, we introduce additional notation.

For a set of strings S, we first generate its suffix tree. We sort the suffix tree such that the leaves follow a lexicographic order from the left side of the tree to the right side. We also construct the *ordered* set of suffixes of S: $\mathsf{O}=\{\mathsf{suf}_1,\cdots,\mathsf{suf}_\mathsf{N}\}$. In this suffix tree, each edge contains at least one character and at most the entire substring ς. The length of the *edge* is the number of characters it contains. If m is the size of the longest string, we define sets $\mathsf{Level_i}$ for $i \in [m]$ that contain the values of edges for each level of the suffix tree. Fig 8 shows an example of a suffix tree, its multiple levels, and edges of different lengths.

To keep up with the same philosophy as SA-ORAM, one could store elements of $\mathsf{Level_i}$ in ORAM$_i$. Yet, to update the suffix tree, it is often required to split an edge into two new edges, see Fig 6. This operation pushes all nodes below this split edge one level downwards. Therefore, if we associate, at the beginning, each edge to a specific level of ORAM recursion, one update degrades the scheme's efficiency. The user has to read all edges below the updated one and rewrite them to ORAM$_i$. This stems from the fact that we associate every edge to a level independently of the number of characters in the edge. Consequently, every creation of a new edge in the middle of the structure implies a translation of the entire subtree below that edge. In conclusion, updating this structure will be very inefficient

A first idea to mitigate this issue is to store any edge with length i into ORAM$_i$ independently of its position in the suffix tree. This modification ensures an oblivious update: whenever an edge is split, it will not affect its position but only trigger the creation of new edges. As described in the suffix tree construction, the split is captured by the two operators, cont and diff. Refer to Fig 6 for an example of suffix' insertion.

ST-ORAM's objective is to associate any node in the suffix tree to a *fixed* level, such that updates will not affect the placement of previously stored nodes. Since any two accesses have to be indistinguishable and the substrings searched for may have different lengths, the server has to always perform m steps.

To build a hierarchical suffix tree over ORAM similar to SA-ORAM, we then have to quantify the number of nodes (or edges) in each level. Suffix trees with N suffixes (leaves) contain up to $2 \cdot N$ nodes in total. With n the number of total strings stored and ct a non-negative integer, our goal is to uniformly distribute the number of nodes over the m levels of ST-ORAM such that each ORAM in the recursive structure contains exactly $\frac{N}{m} \leq ct \cdot n$.

4.3 Suffix Tree Encoding (STE)

We introduce a new *suffix tree encoding* (STE) that enables uniform partitioning of nodes within the recursive ORAM structure. STE fixes an upper bound ct for each ORAM$_i$. This upper bound ensures that during insertions we never exceed the capacity of each ORAM. In practice, ct is small. We will show in Lemma 4.1 that $ct < 3$. To achieve this, our encoding (I) stores leaves based on the suffix length, and (II) interior nodes are distributed following a geometric series with ratio $\frac{1}{2}$. Part (I) of our encoding implies an upper bound of n, and (II) an upper bound of $2 \cdot n$, therefore the maximum number of elements per ORAM equals $3 \cdot n$. Hence, the ct equals 3.

Contrary to k-ary trees, suffix trees do not have a well defined distribution of interior nodes. Some levels of the suffix tree might contain more nodes than others. The distribution of interior nodes in suffix trees depends on the string construction. The idea of our suffix tree encoding (STE) is to give the suffix tree a particular structure to control the distribution of nodes over levels. STE divides a suffix tree in m levels such that each level has $O(n)$ nodes. STE's purpose is to avoid the worst-case scenario where a level might have $O(m \cdot n)$ nodes. For a maximum number of nodes n, STE ensures that any level has an upper bound of nodes that it will not exceed during updates. That is, it avoids overflow scenarios where a level cannot store additional nodes anymore. More formally, we will show that for n strings the number of nodes in each level of ST-ORAM is upper bounded by $3 \cdot n$.

STE uses two encoding rules. The first applies to leaf nodes, and the second to interior nodes:

1. Insert each leaf node l_i of suffix suf_i, for $i \in [N]$ into $\mathsf{Level}_{|\mathsf{suf}_i|}$ of the suffix tree.

2. For each interior node v_i, find child v_j with the shortest distance from the root. Insert v_i in level Level_{j-1}.

Here, the distance is defined as the number of characters of an edge. If two edges have different lengths, they will reside in different levels of the tree. For illustration, Fig 8 depicts the encoding of string "aacaba".

Lemma 4.1. *For n strings encoded with STE, the number of nodes in each level* Level_i *is at most* $3 \cdot n$.

Proof. STE's first rule stores any leaf associated to a particular suffix suf of the suffix tree, into the $|\mathsf{suf}|^{\text{th}}$ level of the new encoded tree. For n strings, we store in the worst case $n \cdot m$ suffixes in total. So, per level Level_i, we need to store n suffixes. This n results from the theoretical upper bound of suffixes per each level. Recall that each level is associated to a string length and therefore we can have up to n suffixes representing all possible suffixes that have a particular length. Thus, it is sufficient to show in the remaining of the proof that the number of interior nodes per level does not exceed $2 \cdot n$.

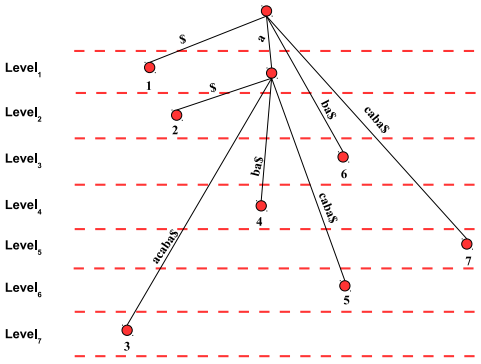

Figure 8: Illustration of suffix tree encoding of string aacaba

We know that there exists a nonnegative integer ϱ such that $\gamma^{\varrho} < n$ and $\gamma^{\varrho+1} > n$ where the upper levels of the tree takes all possible combinations of a suffix tree. Therefore, for $i < \varrho$, we know that the number of interior nodes in Level_i does not exceed n. For $i > \varrho$, recall that for now we have in each level up to n leaves, any interior node has at least 2 children, the worst case for the $(\varrho+1)^{\text{th}}$ level is that each node has to have exactly 2 children where at least one of them is in the $(\varrho+2)^{\text{th}}$. Having more than 2 children will reduce the number of interior nodes since more nodes will map to the same parent. In fact, we show that each level $\varrho < i \leq m$ contains $\sum_{j=0}^{n \log n} \frac{n}{2^j} < 2n$ interior nodes (which is the formula for the geometric series with ratio $\frac{1}{2}$). First, notice that each i^{th} level has n interior nodes which are parent each of 2 leaves. This represents the worst case as explained above. These nodes exist also in the $(i+1)^{\text{th}}$ and they have at most $\frac{n}{2}$ parents in the i^{th} level, recursively, we can show that the size of the interior nodes equals $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 1 = \sum_{j=0}^{\log n} \frac{n}{2^j}$. □

For ST-ORAM with a bounded size n, we start the construction with an empty suffix tree that will be filled based on our STE encoding. For the setup phase, the required parameters are the maximum number of strings, n, the size of the longest string m and the alphabet size γ. Based on n and γ, we generate the non-negative integer ϱ that verifies: $\gamma^{\varrho} < n$ and $\gamma^{\varrho+1} > n$.

4.4 ST-ORAM details

4.4.1 SetupOSS: ST-ORAM setup phase

Create $m-2$ binary tree ORAMs, $\{\text{ORAM}_i\}_{1<i\leq m-1}$, with a number of leaves equal to γ^i for $1<i\leq\varrho$, and $2n$ for $i>\varrho$. We create another tree, denoted ORAM_m, that will contain all the suffixes and then has $m\cdot n$ leaves. Elements of the first level Level_1, which contains only one block, will be stored locally in the user side. Children of the same parent belonging to the same level will be enhanced with a binary search tree-like indexing, i.e., if a parent has γ children in the i^{th} level, for example, one can find the desired child in $\log\gamma$ accesses. That is, instead of storing γ pointers, only two pointers are needed in each block. Thus, the block structure in the i^{th} level contains: an identifier for the ORAM tree, two pointers for the next two children (it can be a leaf) and one alphabetical character, plus, a counter that will guide the binary search. The counter represents the number of characters in common between the children of the same parents. In fact, similarly to SA-ORAM compression technique, redundant information is not needed to lead the search. The alphabetical character, plus, the counter are sufficient to enable the user to decide which pointer to follow within the same level. Note that for each level, during the search, the user has to access $\log\gamma$ times since any parent can have at most γ children. Finally, for occurrence search, two pointers are required to define the beginning and the end of the substring interval, similar to a traditional suffix tree. Thus, the size of any block in the i^{th} level equals: $2\log m\cdot n+3\log n+\log\gamma\cdot\log m=O(\log n\cdot m+\log\gamma)$. The block in ORAM_m contains: an identifier and the entire suffix. That is, the size of blocks in ORAM_m equals $\log n\cdot m+m\log\gamma$. Refer to Fig 9 for a high level description of ST-ORAM setup phase.

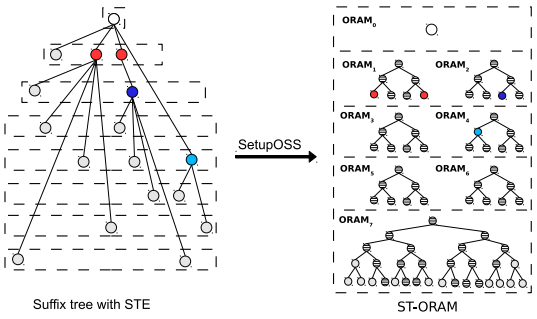

Figure 9: Suffix tree contains two strings with $m=7$, aacaba and bcbbaa, stored using STE. The result includes four interior nodes and twelve leaves.

4.4.2 SubstringQuery: ST-ORAM search phase

For a substring ς, the user locally retrieves from the first block the associated child address. The user generates at random the address and updates the block. Then, the user accesses the first tree ORAM_1 to retrieve the element. $\log\gamma$ accesses have to be performed to stay oblivious even if the child has been found in lesser steps. Also, if the leaf identifier has been found, the user has to make dummy queries to all other left ORAM trees to stay oblivious as well. Otherwise, the user repeats the same steps till finding the leaf identifier. Finally, the user accesses ORAM_m to retrieve the suffix. If ς is within the found suffix then the user outputs true with the number of occurrences occ, or $\perp$ otherwise.

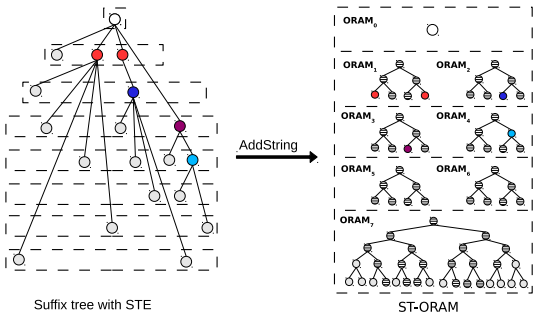

Figure 10: We add the suffix cca. This triggers the creation of a new interior node as well as a new leaf respectively in ORAM_3 and ORAM_7.

4.4.3 AddString: ST-ORAM string insertion

As shown in Fig 6, the insertion of a suffix in the suffix tree results either in the creation of a new leaf or an interior node alongside with a new leaf. In ST-ORAM, the insertion should follow the STE encoding rules. In the following, we are not going to detail how the identifiers or the next leaves identifiers are assigned for each new block since it is very similar to SA-ORAM construction.

Formally, to insert a string s, the user generates all the associated suffixes $\mathsf{Suf(s)}=\{\mathsf{suf_1},\cdots,\mathsf{suf_m}\}$ and performs for each $i\in[m]$:

- Search for $\mathsf{suf_i}$ using ST-ORAM algorithm SubstringQuery, if the suffix already exists, do nothing. Otherwise, if the accessed path p_i has some overlapping between the suffix $\mathsf{suf_i}$, output $\mathsf{pref}=\mathsf{cont}(p_i,\mathsf{suf_i})$ and $\mathsf{diff}(\mathsf{pref},p_i)$. The path p_i is a string defined as the concatenation of data contained in all blocks accessed from ORAM_0 to ORAM_m

during the search of $\mathsf{suf_i}$. Recall that cont and diff are respectively an operator that outputs the common prefix and the inclusive difference between two strings

- Create a new block that has as a prefix pref and has at least two children, one of them is a new substring that will be stored in ORAM_m. The pointers to these two children are respectively: $\mathsf{diff}(\mathsf{pref},p_i)$ and $\mathsf{diff}(\mathsf{pref},\mathsf{suf_i})$. The block will be stored in $\mathsf{Level_j}$, i.e., ORAM_j, such that in the $(j+1)^{\text{th}}$ level, the block has at least one child either from previous nodes or the new substring, this ensures the STE encoding algorithm.

Refer to Fig 10 for a high level explanation of AddString protocol.

5 Analysis

5.1 Security analysis

In this section, we proof security only for ST-ORAM. For SA-ORAM, the proof is similar.

Theorem 5.1. *SA-ORAM is a secure OSS following Definition 2.1, if every bucket is a secure ORAM.*

(Sketch). ST-ORAM is composed of a logarithmic number of ORAMs. Each operation consists of multiple accesses through all the trees. Since, for any operation, all trees are equally accessed, our problem boils down to study only the i^{th} tree access of the ST-ORAM structure. For each of the two operations AddString and QuerySubstring, the i^{th} tree access consists of a number of ORAM operations such that AddString consists of $\log\gamma$ number of ReadAndRemove followed by an Add accesses. Consequently, the operations of ST-ORAM for the i^{th} tree are equivalent to ORAM operations. Thus, if the buckets of the tree are secure ORAMs, the access pattern induced by these operations are indistinguishable based on tree-based ORAM security. This is true for all trees of ST-ORAM and concludes the proof. □

5.2 Theoretical efficiency analysis

We asymptotically compare between our proposed schemes SA-ORAM, ST-ORAM against the oblivious data structure (ODS). For this, SA-ODS and ST-ODS respectively represent the insertion of suffix arrays and suffix trees into the ODS.

First, we should define the construction of a block in SA-ODS and ST-ODS. The block in SA-ODS contains the addresses of two children, a tag as well as the suffix information that can be the entire string. The block in ST-ODS contains up to γ children, a tag as well as the suffix information that can equal the entire string. The block size of SA-ODS $\mathcal{B}_{sa}$ equals $(3\cdot\log N+m\cdot\log\gamma)$ while the block size of ST-ODS $\mathcal{B}_{st}$ equals $((\gamma+1)\cdot\log 2N+m\cdot\log\gamma)$. Thus, the exact communication complexity S equals $\mathsf{S}(\text{SA-ODS})=2z\cdot\log^2 N\cdot\mathcal{B}_{sa}$, and $\mathsf{S}(\text{ST-ODS})=2z\cdot m\cdot\log 2N\cdot\mathcal{B}_{st}$. We recapitulate in the following the respective communication asymptotics for SA-ORAM and ST-ORAM.

Communication overhead SA-ORAM The search is performed by two operations (download/upload) on the entire structure for $1<j\leq l$. For a string length m and an alphabet size γ, the size of each block in SA-ORAM equals $\tau\cdot(\log m\cdot\log\gamma+\log\tau^{j+1})$ for $1<j\leq l-1$. For $j=l$, the size of each block has to be in $\tau\cdot(\log N+m\log\gamma)$. Thus, the total communication complexity S is equal to

$$\begin{aligned}\mathsf{S}(\text{SA-ORAM}) &= 2(z\cdot\tau\cdot(\log N+m\log\gamma)\cdot\log\tau^{l-1}+\\ &\quad\sum_{j=2}^{l-1}z\cdot\tau\cdot\log\tau^{j-1}\cdot(\log m\cdot\log\gamma+\log\tau^{j+1}))\\ &= 2z\cdot\tau\cdot(l-1)\cdot\log\tau\cdot(\log N+m\log\gamma+\\ &\quad\frac{(2\log\tau+\log m\cdot\log\gamma)(l-2)}{2}+\frac{\log\tau\cdot(l-2)(2l-3)}{6})\end{aligned}$$

Communication overhead for ST-ORAM For a search/insert phase, the number of accessed nodes is exactly the same. The communication complexity equals:

$$\begin{aligned}\mathsf{S}(\text{ST-ORAM}) &= 2\sum_{i=2}^{m}z\cdot\log\gamma\cdot\log 2n\cdot(2\log(m\cdot n)+\log\gamma\cdot\log m)+\\ &\quad 2z\cdot\log(m\cdot n)\cdot(\log(m\cdot n)+m\log\gamma)\\ &= 2z\cdot\log\gamma\cdot\log 2n\cdot m\cdot(2\log(m\cdot n)+\log\gamma\cdot\log m)+\\ &\quad 2z\cdot\log(m\cdot n)\cdot(\log(m\cdot n)+m\log\gamma).\end{aligned}$$

In Table 1, we compare the communication complexities of SA-ORAM, ST-ORAM, SA-ODS, and ST-ODS when instantiated with different values of m the length of the string and γ the size of the alphabet. We assume that $z\in O(1)$ which is a result of using Path ORAM in our ORAM_i instantiations in both SA-ORAM and ST-ORAM. In addition, we consider in our asymptotics results that $m\in o(n)$ which means that the length of the strings m is negligible against the number of strings n. If m, $\gamma\in\Omega(\log N)$, SA-ORAM asymptotically improves upon

SA-ODS by a $\log\log n$ factor, and by a $\log n$ factor for m, $\gamma \in \Omega(\log^2 n)$. For $m,\gamma \in \Omega(\log^2 n)$, ST-ORAM improves by a multiplicative factor of $\log\log n \cdot \log n$ over ST-ODS. For constant values of m and γ, asymptotically ST-ORAM and ST-ODS are the most efficient with a smaller hidden constant for ST-ORAM. Also, Table 1 demonstrates that inserting naively a suffix tree in an ODS does not scale well specially for larger values of m and γ. Figures 11, 12 and 13 represent the exact communication in bits for the three different scenarios. We have considered a bucket size z equal to 5 for all the four techniques. The number of considered strings n goes from 10 to 10^{10} elements. We show that SA-ORAM is the most efficient technique that can reduce up to $32\times$ compared SA-ODS and $2000\times$ compared to ST-ODS. Clearly, for static data set scenarios, SA-ORAM outperforms the other schemes. For dynamic scenarios, ST-ORAM can reduce up to $7\times$ compared to SA-ODS and $10\times$ compared to ST-ODS, for constant m and γ. To search obliviously for a substring, the user has to download/upload 103 KBytes in total for 10^{10} stored string. Also, based on Amazon S3 pricing [1], SA-ORAM compared to SA-ODS is up to $2\times$ cheaper for larger $m,\gamma \in \Omega(\log n)$. For example, 1000 accesses cost 0.2 United States Dollar (UDS), while 0.4 USD in SA-ODS for 10^{10} elements, see Fig 14.

5.3 Experimental analysis

For proof of concept, we present our SA-ORAM implementation using Java. The source code is publicly available for download in [12]. This implementation has been done in a client-server setting in order to have realistic measurements of the implementation. The server chosen for our setting is a M3 Amazon EC2 instance running Ubuntu Server 14.04 LTS that has 15 GBytes of RAM and 2 x 40 SSD storage [2], while the client is a 64 bit laptop with 8 cores 2.4 GHz CPU and 8 GByte RAM running Ubuntu 14.04 LTS. To send one ICMP packet to the server, we need 35 ms one way communication as an average of 10 ping commands. The wire bandwidth between the server and the client is 2 Mbits per second.

For our benchmarking, we have chosen the publicly available data set, Google book English 1-gram [16], that contains $\sim$ 10 GBytes of uncompressed strings with a total of $\sim$5 millions strings. The longest string has 100 characters and the alphabet is ASCII, i.e., $m = 100$ and $\gamma = 128$. The n-grams, for $n > 1$, are composed of the 1-gram and will not add any additional new string to the corpus. Therefore, we made the choice to only benchmark our implementation based on 1-grams to reduce the parsing time. Google book 1-grams are based on unique non-redundant strings that enable us to divide the data set to smaller ones for our measurements while keeping a linear division of the number of strings.

Our measurements has been elaborated following two aspects: first, indexing phase that generates the distribution of suffixes given a set of strings, second, a real-time measurements based on the communication overhead and time needed for a substring search. For the real-time measurements, for each input we run 10 times and we take the average of the runs.

We have divided the corpus to four datasets with size respectively equal to 1, 2, 4 and 8 GBytes. From each, we extracted the strings and generated all suffixes. We compare the number of generated suffixes to the upper bound that is nearly equal to the multiplication of number of strings with the size of the longest string. For instance, the 8 GBytes dataset, the longest string had 100 characters long. Fig 15 summarizes the results of suffix extraction. The number of suffixes is $\sim$2 to 3 time larger than the number of generated suffixes. This stems from the fact that English words are not randomly generated and follow a particular distribution where many strings share common suffixes.

Afterwards, we generate the SA-ORAM OSS structure. The size of this structure varies between 2.5 to 10 GBytes for a number of suffixes of 2 to 11 millions. The size of OSS structure has been measured after serialization and it does not provide the optimal solution. The size of the OSS in our case is very language dependent, i.e., the code is based on Java object instantiation and there is an enormous waste of memory which is allocated to objects rather than the actual necessary data. Thus, the size of the OSS can be drastically reduced if the coding is optimized and also if low-level language is used such as C language.

For comparison purpose, we have implemented SA-ODS in Java. The code of the implementation is also publicly available in [12] in a client-server setting. The size of the SA-ODS varies between 5 to 20 GBytes which is twice the value of SA-ORAM and validates therewith our theoretical results. Recall that SA-ODS should store twice the number of leaves compared to SA-ORAM and therefore will create another level in the binary tree of Path ORAM.

Besides, each bucket of the recursive binary trees contains five blocks. The composition of the block is as follows: one leaf tag, one leaf identifier, τ next level identifiers, τ next level leaf tags, plus, the data load. For our experiments, we have fixed τ to 2.

In Fig 16, we provide the average of the communication overhead in KBytes for substring search and compare SA-ORAM and SA-ODS. The standard deviation is based on 10 runs and equals 8 KBytes. The packet's size are

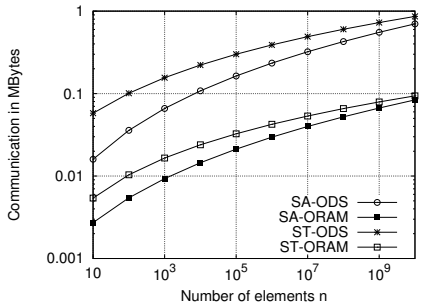

Figure 11: Communication per access for $m,\gamma \in \Omega(1)$

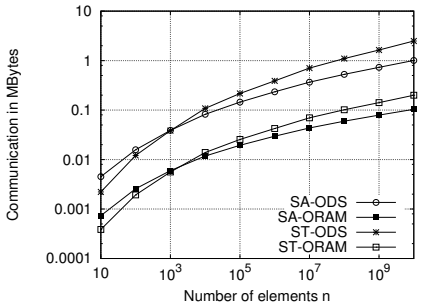

Figure 12: Communication per acces for $m,\gamma \in \Omega(\log n)$

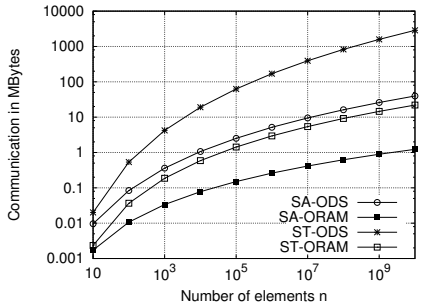

Figure 13: Communication per acces for $m,\gamma \in \Omega(\log^2 n)$

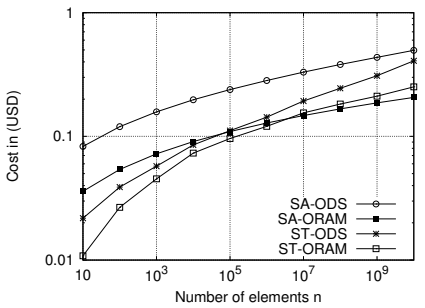

Figure 14: Cost for 1000 accesses for $m,\gamma \in \Omega(\log n)$

for a public network communication between our laptop and Amazon instance. While in our theoretical results, the communication saving of SA-ORAM over SA-ODS is around 7 times, we have only two times saving in practice. This is a result of the network packet construction, we have all the setup communication, the packets headers, plus, the packet's load distribution. Moreover, we have measured in Fig 17 the time of to search for a particular substring in two settings. The first one is in a private network and the second one is in a public one. The average time to perform a search is equal to 640 and 2700 milliseconds for SA-ORAM while 700 to 2900 milliseconds for SA-ODS, with a standard deviation respectively equal to 25 and 60 ms and a relative standard deviation equal to 3.9 and 2.22 for the private and public networks. The time greatly increases for the second setting since the server is outsourced with a reachability equal to 35 ms per packet. Unfortunately, the search in SA-ORAM is an interactive one where the server and the client has to communicate a number of rounds equal to the number of recursive trees generated and this creates some latency issues. For instance, for 11 millions suffixes, 24 recursive trees are created. Therefore, there are 24 rounds of communication between the client and the server.

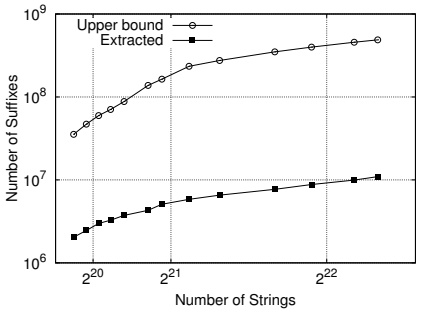

Figure 15: Suffix extraction from the Google 1-gram dataset

6 Related work

Searching over encrypted data can be done with different techniques that offer different levels of security and efficiency. In particular, we focus on techniques that can enable substring search over encrypted data. In the following, we categorize these techniques depending on two main categories: (1) techniques that can enable very fast search but disclose the access pattern, and (2) techniques that can hide the access pattern but induce more overhead.

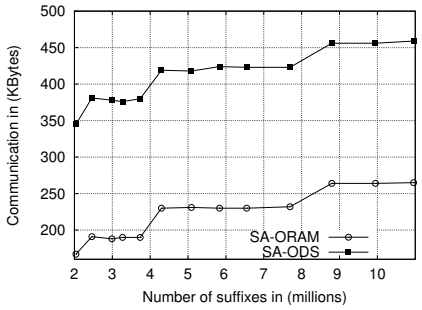

Figure 16: Communication overhead in KBytes

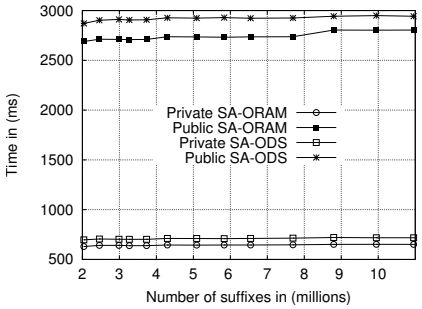

Figure 17: Time measurements in private and public network

6.1 Searchable Encryption

Substring search can be achieved by any symmetric searchable encryption (SSE) scheme. SSE has been introduced by Song et al. [26] and has been improved to achieve sublinear search [7, 9]. Many works target several search functionalities, such as similarity, multiple keyword, range, fuzzy search and the possibility to update over encrypted data [3–6, 18, 21]. SSE can be adapted to substring search by using straightforward solutions such as generating all possible substrings. These substrings can then be considered as the keywords and securely be stored using a sublinear SSE. However, this solution induces a lot of storage overhead since the user has to generate all possible combinations of substrings of any string. For example, for an alphabet with a size γ and a string length m, the number of possible substring equals γ^m which is clearly infeasible for real-life values such as $m{=}\gamma{=}2^7$.

Recently, Chase and Shen [8] introduced a secure substring search based on suffix trees in $O(\lambda{\cdot}\varsigma{+}\mathsf{occ})$ to search for a substring ς occurring occ times, where λ is the security parameter. Trivial solutions based on SSE are more efficient, but the scheme [8] greatly enhances the storage efficiency to be in $O(N)$, where N is the number of possible suffixes of all strings.

6.2 Oblivious RAM

Oblivious RAM (ORAM) has been introduced by Goldreich and Ostrovsky [15]. ORAM can enable a user to outsource his data encrypted and later on query the server without disclosing which data has been retrieved. The main ORAM issue was its linear shuffling that has been solved recently by introducing tree-based ORAM by Shi et al. [25]. Tree-based ORAMs offer poly-logarithmic worst case that has been substantially improved by [10, 11, 27]. ORAM can be applied, as SSE, to solve substring problem. Clearly, the trivial solution that generates all possible substrings is clearly not feasible. Recently, Wang et al. [30] have adapted Path ORAM [27] to store any tree or graph structure obliviously by inducing a $\log N{\cdot}\nu$ blow up compared to plaintext solutions, where ν is the factor of block size increase. This oblivious data structure (ODS) can be adapted to store suffix arrays, suffix trees and therefore enable oblivious substring search.

There are many other techniques that can also be applied to solve secure substring search, such as functional, predicate or homomorphic encryption [13, 19, 20] or private information retrieval [24]. However, these techniques will be impractical in practice, and there are other ways to solve the problem with better efficiency.

Our objective in this paper is to present the first efficient technique that solves obliviously substring matching. For this, we make use of tree-based ORAM as a building block and we particularly adapt it to handle in a more efficient way the substring search.

SA-ORAM and ST-ORAM in SSE: Using SA-ORAM or ST-ORAM, we can provide a symmetric substring search over encrypted data. To this end, we should store the mapping between the substrings and the strings and also eventually the mapping between the substrings and the corresponding records. For instance, an investigation aims to search for all URL logs that contain "facebook" and find out the employees that have accessed these websites. Also, the investigators should make sure that the matching URL logs is exactly for the banned websites. Thus, there will be a need to retrieve the entire URL logs to avoid uncertainty. For example, the employee has accessed "www.facebooking.com" which is totally a different website, but his URL logs has been retrieved during the substring search phase.

To use SA-ORAM in a SSE scenario, we can utilize any secure inverted index for this purpose, such as [7, 9]. We can store strings and records identifiers in the secure inverted indexes. Finally, the pointer of each secure index

entry should has to be added in the last level of SA-ORAM that will map each suffix to *all* records/strings that contain it.

However, we should emphasize that disclosing the mapping between the substrings and the strings may have severe consequences. If an adversary has an entire knowledge of the dictionary, it can easily construct a frequency table that associates each substring to the actual keywords in the dictionary. In our construction during the search, even if we hide which substring is retrieved, the retrieved mapping eventually discloses the number of association, i.e., the size of the set $\{a_1,\cdots,a_2\}$ is enough to give valuable information to the server. An eventual solution yet expensive to this issue might be an ORAM structure that stores these indexes. In this paper, providing security model for SA-ORAM over SSE is out of our scope.

7 Conclusion

We have presented the first two techniques specifically for oblivious substring search over encrypted data. Our oblivious suffix arrays construction SA-ORAM provides best for static string set. Our second construction, ST-ORAM can handle updates, but at the additional cost of a (small) multiplicative factor. Both of our constructions asymptotically improve naive solutions by a $\log n$ factor. Quantitatively, we have shown that SA-ORAM results in up to $32\times$ less communication overhead compared to suffix arrays over oblivious data structure [30]. As future work, we aim to investigate the extension of the obliviousness feature on more search types such as range or fuzzy search.

References

[1] Amazon. Amazon s3 pricing, 2014. `http://aws.amazon.com/s3/pricing/`.

[2] Amazon. Amazon EC2 Pricing, 2015. `http://aws.amazon.com/ec2/pricing/`.

[3] Alexandra Boldyreva and Nathan Chenette. Efficient fuzzy search on encrypted data. *IACR Cryptology ePrint Archive*, 2014:235, 2014.

[4] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 578–595, 2011.

[5] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael 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.

[6] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*, 2014.

[7] Melissa Chase and Seny 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.

[8] Melissa Chase and Emily Shen. Pattern matching encryption. *IACR Cryptology ePrint Archive*, 2014: 638, 2014.

[9] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 79–88, 2006.

[10] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, and Ling Ren. Onion ORAM: A constant bandwidth and constant client storage ORAM (without FHE or SWHE). *IACR Cryptology ePrint Archive*, 2015:5, 2015.

[11] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A Low-Latency, Low-Area Hardware ORAM Controller with Integrity Verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[12] Anonymized for submission. SA-ORAM source code, 2015. `https://www.dropbox.com/s/oubkacv3ndxh1im/SAORAM.zip?dl=0`.

[13] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM*

Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009, pages 169–178, 2009.

[14] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Proceedings of Privacy Enhancing Technologies*, pages 1–18, 2013.

[15] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996. ISSN 0004-5411. doi: 10.1145/233551.233553. URL `http://doi.acm.org/10.1145/233551.233553`.

[16] Google. Google Books Ngram Viewer, 2012. `http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`.

[17] D. Gross. 50 million compromised in Evernote hack, 2013. `http://www.cnn.com/2013/03/04/tech/web/evernote-hacked/`.

[18] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 310–320, 2014.

[19] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Advances in Cryptology–EUROCRYPT 2008*, pages 146–162. Springer, 2008.

[20] Allison Lewko, Tatsuaki Okamoto, Amit Sahai, Katsuyuki Takashima, and Brent Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *Advances in Cryptology–EUROCRYPT 2010*, pages 62–91. Springer, 2010.

[21] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5. IEEE, 2010.

[22] Udi Manber and Gene 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.

[23] R. Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the Symposium on Theory of Computing –STOC*, pages 514–523, Baltimore, USA, 1990.

[24] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *Proceedings of the Symposium on Theory of Computing –STOC*, pages 294–303, El Paso, USA, 1997.

[25] E. Shi, T.-H.H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost. In *Proceedings of Advances in Cryptology – ASIACRYPT*, pages 197–214, Seoul, South Korea, 2011. ISBN 978-3-642-25384-3.

[26] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 44–55, 2000.

[27] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM Conference on Computer and Communications Security*, pages 299–310, 2013.

[28] Theguardian. The Interview revenge hack cost Sony just $15m , 2015. `http://www.theguardian.com/film/2015/feb/04/guardians-peace-revenge-hack-sony-finances-unscathed/`.

[29] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[30] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 215–226, 2014.

[31] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973.