Secure Compressed Suffix Arrays

Kunihiko Sadakane
The University of Tokyo, Japan

Abstract

This paper proposes a *secure compressed suffix array*, which is a data oblivious and compressed version of the suffix array used for finding substrings of a large string. Secure compressed suffix arrays can be used for indexing a large collection of strings containing personal information such as DNA data.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases suffix array, compression, encryption, oblivious algorithm, secure computation

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.13

Category Research

1 Introduction

As the amount of data increases, the following points are becoming more problematic.

- Processing time for analyzing the data increases. We need efficient algorithms and data structures.
- We need to care about the use of sensitive data such as personal information.

For the former problem, many researches on algorithms for compressed data have been conducted. Seminal results on this topic are succinct bit-vectors [6, 11], succinct ordered trees [9], and compressed suffix arrays [4, 2]. For the latter problem, *data anonymization* [16] and *secure computation* [17] have been proposed. Because data anonymization modifies input data, some information will be lost. We focus on secure computation, which is a technique to process encrypted data without decryption.

There are two main schemes for secure computation: secret sharing [13] and fully homomorphic encryption [3].

Assume that there are a client and a server. There are two main scenarios.

1. A client has private data and wants to use a cloud service to process the data. Data are stored in the cloud server in an encrypted form. The client runs a program on itself. When some data are necessary, the client asks the server to obtain the data. Then the client decrypts the data, does some computation, encrypts the data and sends back to the server. In this scenario, the task of the server is to store the data and give accesses to a part of the data to the client. The data must be stored as an encrypted form on the server, but computation on encrypted data is not necessary. Therefore it is enough to hide the access pattern to the data on the server.
2. The server stores encrypted data and the secret key is not known to the server. The client asks the server to run a program. The server does some computation and returns the answer to the client. The client decrypts the answer using the secret key. In this scenario, algorithms executed on the server must be data oblivious and the computation must be done on encrypted data.

The second scenario is preferable because the client does not require computation power for analyzing big data. However, we need to design special algorithms for the server which are data oblivious and which run on encrypted data. We call such algorithms *secure algorithms*.

This paper proposes *secure compressed suffix array*, which is a data oblivious and encrypted version of the compressed suffix array [4].

2 Preliminaries

2.1 Suffix arrays

Let T be a string of length n on alphabet $\mathcal{A}$ of size σ. The j-th character of T is denoted by $T[j]$ ($j = 0, 1, \ldots, n-1$) and it belongs to $\mathcal{A}$. We assume that a unique terminator \$, which is smaller than any character in $\mathcal{A}$, is appended to T. That is, $T[n] = \$$. A substring $T[i]T[i+1]\cdots T[j]$ of T is denoted by $T[i, j]$. The substring $T[j, n]$ is called the j-th suffix of T and denoted by T_j.

The suffix array [8] of the string T is an integer array $\mathrm{SA}[0, n]$ defined as $\mathrm{SA}[i] = j$ ($i = 0, 1, \ldots, n$) if T_j is the lexicographically the i-th suffix of T. It always holds that $\mathrm{SA}[0] = n$, which corresponds to the shortest suffix consisting of only the terminator.

If we have T and SA, we can support the following queries:

- $\mathrm{Count}(P, T)$: returns the number of occurrences of P in T in $\mathrm{O}(|P| \log n)$ time
- $\mathrm{Locate}(P, T)$: returns the positions of occurrences of P in T in $\mathrm{O}(|P| \log n + \mathit{occ})$ time where $\mathit{occ} = \mathrm{Count}(P, T)$

To support these queries, we use the following query:

- $\mathrm{Range}(P, T)$: returns the maximal range $[s, e] \subset [0, n]$ so that for any $i \in [s, e]$ the substring $T[SA[i], SA[i] + |P| - 1]$ is equal to P.

Let $[s, e] = \mathrm{Range}(P, T)$. Then it holds $\mathrm{Count}(P, T) = e - s + 1$ and $\mathrm{Locate}(P, T) = \{\mathrm{SA}[s], \mathrm{SA}[s+1], \ldots, \mathrm{SA}[e]\}$.

The size of the suffix array SA is $n \log n$ bits. We also need to store the string T itself, which occupies $n \log \sigma$ bits. The suffix array requires a huge space compared with the string itself, especially for strings on small alphabets, such as DNA strings. For human DNA, $\sigma = 4$, whereas $n > 2^{31}$. Then $\log n$ is more than 15 times larger than $\log \sigma$.

2.2 Succinct bit-vectors

Succinct data structures are data structures storing objects in minimum number of bits. More precisely, consider storing an element x of a set S. Then the *information-theoretic lower bound* of the number of bits to represent x is defined as $\lceil \log_2 x \rceil$. Hereafter we omit the base 2 of logarithm. Let Z denote this value. Then a data structure for storing x is called *succinct* if it uses $Z + \mathrm{o}(Z)$ bits, and *compact* if it uses $\mathrm{O}(Z)$ bits.

The most fundamental succinct data structure is a bit-vector supporting rank and select queries. A bit-vector is a string $B[1, n]$ on the binary alphabet $\{0, 1\}$ and rank and select are defined as follows.

- $\mathrm{rank}_c(B, i)$ returns the number of c's in $B[0, i]$. We define $\mathrm{rank}_c(B, 0) = 0$ and $\mathrm{rank}_c(B, i) = \mathrm{rank}_c(B, n)$ if $i > n$.
- $\mathrm{select}_c(B, j)$ returns the position of the j-th c in B. We define $\mathrm{select}_c(B, 0) = 0$ and $\mathrm{select}_c(B, j) = n + 1$ if $j > \mathrm{rank}_c(B, n)$.

For a bit-vector of length n, rankand selectare computed in constant time using the bit-vector itself and an $\mathrm{O}(n \log \log n / \log n)$ bit auxiliary data structure [11].

We briefly review the rank data structure. The bit-vector B is partitioned into super-blocks of length L_1 each, and each super-block is further partitioned into blocks of length L_2 each. We store rank values for all super-block boundaries in array R_1 using $\mathrm{O}((n \log n)/L_1)$ bits, and rank values for all block boundaries in array R_2 using $\mathrm{O}((n \log L_1)/L_2)$ bits. Inside a block, we count the number of c's using table lookups for every $1/2 \log n$ bits. The size of the table is $\mathrm{O}(\sqrt{n} \log n \log \log n) = \mathrm{o}(n)$ and the time complexity is $\mathrm{O}(L_2 / \log n)$. If we set $L_1 = \Theta(\log^2 n)$ and $L_2 = \frac{1}{2} \log n$, we obtain the desired bound.

Table 1 Oblivious RAM data structures for storing n bits. Bits are grouped into blocks of $b \geq \log n$ bits each. Bandwidh blowup is the number of blocks to be accessed to obtain one block obliviously.

Server space (#bits)	Bandwidth blowup	User space (#blocks)	Reference
$\mathrm{O}(n)$	$\mathrm{O}(\log^2 n/\log\log n)$	$\mathrm{O}(1)$	Kushilevitz et al. [7]
$\mathrm{O}(n)$	$\mathrm{O}(\log^2 n)$	$\omega(\log n)$	Stefanov et al. [15]
$n(1+\Theta(1/\log n))$	$\mathrm{O}(\log^2 n)$	$\mathrm{O}(\log n \log\log n)$	Onodera, Shibuya [10]

2.3 Secure algorithms

If we forget about time efficiency, any computation can be done on encrypted data if we can support additions and multiplications on two encrypted integers. There exist two such schemes.

- Secret Sharing [13]. Data are distributed into two or more servers and for each server the stored data look like random values and no information is leaked. Additions can be done locally in each server, whereas for multiplications the servers must comminicate each other.
- Fully Homomorphic Encryption (FHE) [3, 1]. Any number of additions and multiplications on encrypted numbers can be done.

Both schemes have a drawback that the computation is much slower than plain (unencrypted) data. In Secret Sharing schemes the servers communicate each other for computing multiplications. This takes much more time than the computation on plain data in a single server. In FHE schemes, there are no communication because there is only one server. However multiplications are extremely slow. Therefore it is important to develop efficient secure algorithms.

In this paper, we assume that in both schemes, integer addition, multiplication, division, less-than comparison are done efficiently in unit time. Then our algorithm runs in both schemes.

2.4 Oblivious RAM

Oblivious RAMs [7, 15, 10], or ORAMs for short, are data structures supporting oblivious read and write to an array. Without loss of generality we can assume the array stores a binary string S of length n. S can be regarded as an array of length n/w storing w-bit integers. An ORAM has a parameter b called the block size. S is partitioned into blocks of length b each, and a block is accessed as a unit. To achieve oblivious access, more than one blocks are accessed to obtain one block. The ratio is called bandwidth blowup.

Obliviousness is defined as follows. Let $\vec{y} = ((\mathrm{op}_1, a_1, d_1), (\mathrm{op}_1, a_1, d_1), \ldots, (\mathrm{op}_M, a_M, d_M))$ be a sequence of accesses to an ORAM where op_i is either read or write, a_i is the address of the i-th access, and d_i is the content to be written in the a_i-th block when op_i is write. Let $A(\vec{y})$ be denote the sequence of accesses to the server given $\vec{y}$. Then the ORAM is said to be oblivious if two any access sequences $\vec{y}$ and $\vec{z}$ of the same length, $A(\vec{y})$ and $A(\vec{z})$ are computationally indistinguishable by anyone but the client, and if the failure probability is negligible [15].

Table 1 shows existing oblivious RAM data structures. Onodera and Shibuya [10] give a succinct oblivious RAM, that is, the server space is $n + \mathrm{o}(n)$ bits. The other two data structures are compact, that is, the server space is $\mathrm{O}(n)$ bits.

3 Compressed Suffix Arrays

3.1 The original structure

Grossi and Vitter [5, 4] have proposed *compressed suffix arrays*, which are a compressed version of suffix arrays. Let SA be the suffix array of string T of length n on alphabet $\mathcal{A}$ of size σ. The compressed suffix array is a data structure which efficiently supports the following operation:

- Lookup(i): returns SA[i].
- Inverse(j): returns i such that $j = \mathrm{SA}[i]$ (the inverse suffix array).

The core component of the compressed suffix arrays is the Ψ function, defined as follows.

$$\Psi[i] = \mathrm{SA}^{-1}[SA[i] + 1]$$

if $i > 0$, and $\Psi[0] = \mathrm{SA}^{-1}[1]$. The Ψ function has a good property that it is piecewise monotone. Precisely, let $[s_c, e_c]$ be the range of the suffix array such that $T[SA[i]] = c$ for any $i \in [s_c, e_c]$ where $c \in \mathcal{A}$. Then if $i, j \in [s_c, e_c]$ and $i < j$, it holds $\Psi[i] < \Psi[j]$. Then the following function is strictly increasing.

$$\Psi'[i] = T[\mathrm{SA}[i]] \cdot (n + 1) + \Psi[i]$$

We can compress Ψ' in $n(2 + \log \sigma) + \mathrm{o}(n)$ bits so that any $\Psi'[i]$ is computed in constant time. From $\Psi'[i]$, we can obtain $T[\mathrm{SA}[i]]$ and $\Psi[i]$ easily in constant time.

The encoding of Ψ' is as follows. each entry of Ψ' is a $(\log \sigma + \log n)$-bit integer. We partition it into higher part and lower part. The higher part has $\log n$ bits and the lower part has the rest ($\log \sigma$ bits). The lower parts for all entries are stored in an integer array $L[0, n]$ in $n \log \sigma$ bits. The upper parts for all entries are represented by a bit-vector $H[0, 2n]$ as follows. Let $d_i = (\Psi'[i] \div \sigma) - (\Psi'[i-1] \div \sigma)$ for $i = 0, 1, \ldots, n$ (we assume $\Psi'[-1] = 0$). Because Ψ' is increasing, $d_i \geq 0$ for any i. We encode d_i's by unary codes. That is, we write d_i many zeros followed by a one, to H. Then, $H[i]$ can be computed in constant time as

$$H[i] = \mathrm{select}_1(H, i) - i$$

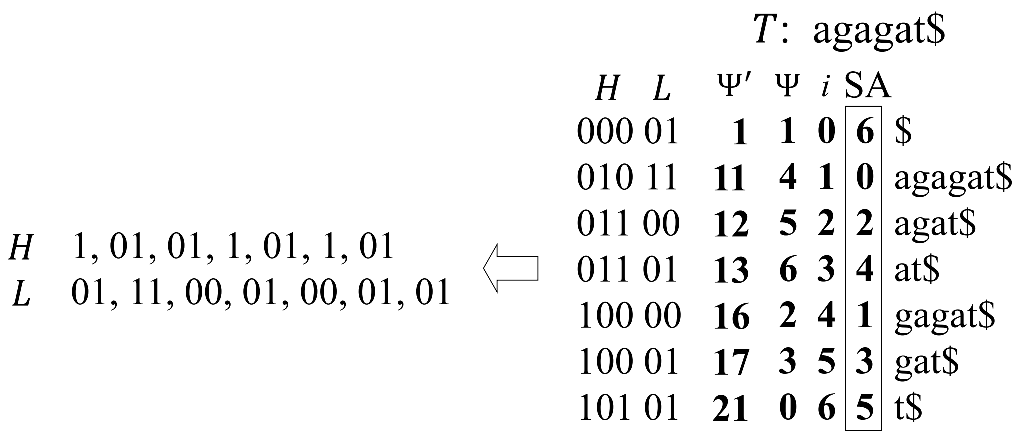
From the definition of Ψ, if $\mathrm{SA}[i] = j$, it holds $\mathrm{SA}[\Psi[i]] = j + 1$. That is, if we know the lexicographic order i of a suffix T_j, we can obtain that of the next suffix T_{j+1} by computing $\Psi[i]$. We sample the values of the suffix array for every L_3 entries and stores them in an array A. We also store a bit-vector $F[0, n]$ such that $F[i] = 1$ iff $SA[i]$ is sampled. Then Lookup(i) is computed as follows.

$$\mathrm{Lookup}(i) = \begin{cases} A[\mathrm{rank}_1(F, i)] & \text{if } F[i] = 1 \\ \mathrm{Lookup}(\Psi[i]) - 1 & \text{if } F[i] = 0 \end{cases}$$

The array A uses $\mathrm{O}((n \log n)/L_3)$ bits and F uses $n + \mathrm{o}(n)$ bits. If we set $L_3 = \Theta(\log n)$, the space for A is $\mathrm{O}(n)$ bits and Lookup(i) is computed in $\mathrm{O}(\log n)$ time. See Figure 1 for an example of

3.2 Self-indexes

The original compressed suffix array is a compressed representation of the suffix array and used together with the string itself. We can change it to a self-index, that is, a data structure for string matching which does not use the string. It is enough to add $n + \mathrm{o}(n) + \mathrm{O}(\sigma \log n)$ bits. We use a bit-vector $D[0, n]$ showing that if $D[i] = 1$ then $i = 0$ or $T[SA[i]] \neq T[SA[i-1]]$.



T: agagat$

H	L	Ψ'	Ψ	i	SA	
000	01	**1**	**1**	**0**	**6**	$
010	11	**11**	**4**	**1**	**0**	agagat$
011	00	**12**	**5**	**2**	**2**	agat$
011	01	**13**	**6**	**3**	**4**	at$
100	00	**16**	**2**	**4**	**1**	gagat$
100	01	**17**	**3**	**5**	**3**	gat$
101	01	**21**	**0**	**6**	**5**	t$

H	1, 01, 01, 1, 01, 1, 01
L	01, 11, 00, 01, 00, 01, 01

Figure 1 An example of compressed suffix arrays.

That is, $D[i] = 1$ iff lexicographically the i-th suffix has a different head character from the $(i-1)$-st suffix. We use an array C of length σ such that $C[\text{rank}_1(D,i)] = T[SA[i]]$. We define $\text{Head}(i) = C[\text{rank}_1(D,i)]$. The array uses $\sigma \log \sigma = \mathrm{O}(\sigma \log n)$ bits (we assume $\sigma \le n$). We use another array $C^{-1}[1,\sigma]$ storing in $C[c]$ the range $[s_c, e_c] \subset [0,n]$ such that for any $i \in [s_c, e_c]$ it holds $T[SA[i]] = c$. This array uses $\mathrm{O}(\sigma \log n)$ bits.

Using C, we can recover a substring of T. Assume the lexicographic order i of a suffix T_j is known ($\text{SA}[i] = j$). Then the character $T[j+k]$ is equal to $\text{Head}(\Psi^k[i])$. The substring $T[j, j+\ell-1]$ is computed in $\mathrm{O}(\ell)$ time by iteratively computing $i := \Psi[i]$. Computing the lexicographic order i of T_j is equivalent to computing $\text{Inverse}(j)$, and it is done similarly to computing an entry of the suffix array using Ψ and the sampled suffix array [12].

To support $\text{Range}(P,T)$, we use what we call *backward search*. Let m be the length of P. First we obtain the range for the shortest suffix $P[m,m]$ by $[s,e] := C^{-1}[P[m]]$. Then, given the range $[s,e]$ for a suffix $P[j+1,m]$, we compute the range $[s',e']$ for the suffix $P[j,m]$. This is done as follows.

$$\begin{aligned} s' &:= \text{argmin}_{i \in C^{-1}[P[j]]} \{\Psi[i] \ge s\} \\ e' &:= \text{argmax}_{i \in C^{-1}[P[j]]} \{\Psi[i] \le e\} \end{aligned}$$

By a simple binary search, the new range is obtained in $\mathrm{O}(\log n)$ time, and therefore $\text{Range}(P,T)$ is done in $\mathrm{O}(|P| \log n)$ time, which is the same as the suffix array [12].

We can simplify the algorithm as follows.

$$\begin{aligned} s' &:= \text{argmin}_{i \in [0,n]} \{\Psi'[i] \ge P[j] \cdot (n+1) + s\} \\ e' &:= \text{argmax}_{i \in [0,n]} \{\Psi'[i] \le P[j] \cdot (n+1) + e\} \end{aligned}$$

Now we do not need the array C^{-1}. This has another merit that it is easier to make it oblivious, which will be shown in the next section.

4 Secure Compressed Suffix Arrays

We show that the compressed suffix arrays can be modified so that all the operations Ψ, Range, Lookup, and Inverse. First we show that Range, Lookup, and Inverse can be done obliviously if Ψ' is computed obliviously. We assume that the lengths of the string T and the pattern P, and the alphabet size σ are public.

4.1 Computing Range

As shown in Section 3.2, $\mathrm{Range}(P, T)$ is done by $|P|$ many binary searches on Ψ'. Given the range $[s, e]$ for $P[j+1, m]$, we compute the range $[s', e']$ for $P[j, m]$. To do so, we first compute $P[j] \cdot (n+1) + s$ and $P[j] \cdot (n+1) + e$, which can be done obliviously. Then using a binary search on Ψ' we compute s'. The initial range is $[0, n]$, and in each step we update the range based on the result of a less-than comparison. Using Ψ' is easier than using Ψ because we do not need the array C^{-1} that must support oblivious accesses. We can compute $\mathrm{Range}(P, T)$ using $\mathrm{O}(|P| \log n)$ many oblivious accesses to Ψ'.

4.2 Computing Lookup and Inverse

$\mathrm{Lookup}(i)$ can be done using the sampled array A, the bit-vector F, and Ψ. The original algorithm repeats computing $i := \Psi[i]$ until $F[i] = 1$. This is not oblivious because the number k of iteration depends on i. Precisely, it holds $\mathrm{Lookup}(i) = A[\mathrm{rank}_1(F, \Psi^k[i])] - k$ where $k \geq 0$ is the smallest number such that $F[\Psi^k[i]] = 1$.

To change the algorithm oblivious, we fix the number of iterations to L_3. Because the suffix array entries are sampled every L_3 entries in text order, for exactly one entry it holds $F[\Psi^k[i]] = 1$ among those for $k = 0, 1, \ldots, L_3 - 1$. Therefore we change the algorithm as follows.

```
1. $x := 0$
2. for $k = 0, 1, \ldots, L_3 - 1$
3.     $x := x + F[i] \cdot (A[\mathrm{rank}_1(F, i)] - k)$
4.     $i := \Psi[i]$
5. return $x$
```

Here we need oblivious accesses to F and A, and oblivious computation of $\mathrm{rank}_1(F, i)$.

For A, we use the Path ORAM [15] with block size $b = \log^2 n$. Then the space usage is $\mathrm{O}(n)$ bits and an entry of A is obtained in $\mathrm{O}(\log^2 n)$ many accesses to blocks, which can be done in $\mathrm{O}(\log^3 n)$ time.

For F, we use the succinct ORAM [10] with block size $b = \log^2 n$. Then a block of F is obtained in in $\mathrm{O}(\log^2 n)$ many accesses to blocks, which can be also done in $\mathrm{O}(\log^3 n)$ time. For computing a rank on F, we use the array R_1 defined in Section 2.2. This is stored using the Path ORAM. The space usage is $\mathrm{O}(n/\log n)$ bits. The rank inside a block is computed by logical operations in $\mathrm{O}(\log n)$ time. Therefore a rank value is computed in $\mathrm{O}(\log^3 n)$ time.

The computation of $\mathrm{Inverse}(j)$ is similar.

4.3 Computing Ψ

Finally, we show how to compute $\Psi[i]$. As shown in Section 3.1, Ψ' is represented by a bit-vector $H[0, 2n]$ and an array $L[0, n]$ of $\log \sigma$-bit integers. The array L is stored using the succinct ORAMs. The bit-vector H is stored similarly to F, but here we also need to store the auxiliary data structure for select_1. This can be stored using the Path ORAM.

To sum up, $\Psi[i]$ is computed in $\mathrm{O}(\log^3 n)$ time and the space usage is $n(2+\log\sigma)+\mathrm{o}(n)\log\sigma$ bits.

4.4 Summary

$\Psi[i]$ takes $\mathrm{O}(\log^3 n)$ time. A step of a backward search is a binary search on Ψ', and therefore it is done in $\mathrm{O}(\log^4 n)$ time. Then $\mathrm{Range}(P,T)$ takes $\mathrm{O}(|P|\log^4 n)$ time. For $\mathrm{Lookup}(i)$, we set $L_3 = \mathrm{O}(\log n)$. Then it takes $\mathrm{O}(\log^4 n)$ time. The space usage is $(n+\mathrm{o}(n))\log\sigma + \mathrm{O}(n)$ bits in total.

5 Concluding Remarks

We have proposed secure compressed suffix arrays. For a string of length n on an alphabet of size σ, It uses $(n+\mathrm{o}(n))\log\sigma+\mathrm{O}(n)$ bits of space and the $\mathrm{Count}(P,T)$ query is done in $\mathrm{O}(|P|\log^4 n)$ time, and the $\mathrm{Lookup}(i)$ query is done in $\mathrm{O}(\log^4 n)$ time. Therefore there is an $\mathrm{O}(\log^3 n)$ multiplicative overhead compared with the original compressed suffix arrays [4]. To improve the running time, we need more efficient succinct ORAM [10] and standard ORAMs [14]. Future work will be developing such oblivious RAM data structures and giving efficient implementions.

References

1 Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, January 2020. doi:10.1007/s00145-019-09319-x.

2 P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

3 Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1536414.1536440.

4 R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

5 Roberto Grossi and Jeffrey S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on the Theory of Computing*, pages 397–406, Portland, OR, 2000.

6 G. Jacobson. Space-efficient Static Trees and Graphs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

7 Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the 2012 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 143–156, 2012. doi:10.1137/1.9781611973099.13.

8 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

9 J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

10 Taku Onodera and Tetsuo Shibuya. Succinct Oblivious RAM. In Rolf Niedermeier and Brigitte Vallée, editors, *35th Symposium on Theoretical Aspects of Computer Science (STACS 2018)*, volume 96 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2018.52.

11 R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007. doi:10.1145/1290672.1290680.

12 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003. doi:10.1016/S0196-6774(03)00087-7.

13 Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. doi:10.1145/359168.359176.

14 Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 197–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25385-0_11.

15 Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2508859.2516660.

16 Latanya Sweeney. k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, October 2002. doi:10.1142/S0218488502001648.

17 Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986. doi:10.1109/SFCS.1986.25.