Succinct Oblivious RAM

Taku Onodera
Human Genome Center, Institute of Medical Science, The University of Tokyo, Tokyo, Japan

Tetsuo Shibuya
Human Genome Center, Institute of Medical Science, The University of Tokyo, Tokyo, Japan

Abstract

As online storage services become increasingly common, it is important that users' private information is protected from database access pattern analyses. Oblivious RAM (ORAM) is a cryptographic primitive that enables users to perform arbitrary database accesses without revealing any information about the access pattern to the server. Previous ORAM studies focused mostly on reducing the access overhead. Consequently, the access overhead of the state-of-the-art ORAM constructions are almost at practical levels in certain application scenarios such as secure processors. However, we assume that the server space usage could become a new important issue in the coming big-data era. To enable large-scale computation in security-aware settings, it is necessary to rethink the ORAM server space cost using big-data standards.

In this paper, we introduce "succinctness" as a theoretically tractable and practically relevant criterion of the ORAM server space efficiency in the big-data era. We, then, propose two succinct ORAM constructions that also exhibit state-of-the-art performance in terms of the bandwidth blowup and the user space. We also give non-asymptotic analyses and simulation results which indicate that the proposed ORAM constructions are practically effective.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols, Security and privacy → Management and querying of encrypted data

Keywords and phrases Oblivious RAM, Succinct Data Structure, Balls-into-bins

Digital Object Identifier 10.4230/LIPIcs.STACS.2018.52

Funding This work was supported by JSPS KAKENHI Grant Number 17H01693, 17K20023JST and CREST Grant Number JPMJCR1402.

Acknowledgements We thank Paul Sheridan for helpful discussion.

1 Introduction

Oblivious RAM (ORAM) is a cryptographic primitive that enables users to access a database on a server without revealing the access pattern to the server. Though originally introduced for software protection [14], ORAM is directly relevant to the present cloud computing.

In the previous studies on ORAM, researchers focused mainly on reducing the access bandwidth cost, a performance measure used as a proxy of the access time. This is because even the current most state-of-the-art ORAM constructions have two or three orders of magnitude larger bandwidth cost than the ordinary (non-secure) accesses. However, in certain settings, the ORAM access is already rather efficient. For example, Maas et al. proposed PHANTOM [23], an ORAM-based secure processor, and reported that if PHANTOM is deployed on the server, SQLite queries can be performed without revealing the access pattern at the cost of 1.2–6× slowdown compared to non-secure SQLite queries. In such cases, it is reasonable to pay more attention to performance measures other than the access speed.

In particular, the *server space usage* is a very important performance measure for big-data applications. First, there are applications where the amount of data is virtually unbounded, and thus the limit of the available space defines the limit of the analyses. Second, due to the cache effect, small memory usage often leads to faster computation. Third, space costs money, especially in a cloud computing server. The second and the third points are especially relevant if the data is meant to be stored in the main memory (by default), which is exactly the case in ORAM application scenarios such as PHANTOM.

In most modern ORAM constructions, if the size of the original database is n bits, the amount of the space required by the server is $n + \Theta(n)$ bits. In this paper, we investigate the possibility of ORAM constructions that need only $n + o(n)$ bits of server space. We call such ORAM constructions *succinct*. This space efficiency formalization is widely used in the field of *succinct data structures* and has proved to be useful to design practically relevant space-efficient data structures in theoretically clean ways.

The main difficulty to achieve succinctness is that most existing ORAM construction approaches rely on the use of linear amount of "dummy" data. The situation is similar to conventional hash tables, which need extra space linear to the stored keys size. Although it seems possible to reduce the constant factor of the extra space to some extent, it is not at all trivial if one can achieve sublinear extra space maintaining the state-of-the-art performance in other aspects such as access bandwidth and user space usage.

Results. Table 1 shows the performance of the proposed methods and the existing methods. Our first construction takes $n(1 + \Theta(\frac{\log n}{B} + \frac{g(n)}{f_1(n)/\log n}))$-bit server space where n is the database size, $f_1(\cdot)$ is a function such that $f_1(n) = \omega(\log n)$ and $O(\log^2 n)$, $g(\cdot)$ is a function such that $g(n) = \omega(1)$ and $o(\sqrt{f_1(n)/\log n})$, and B is the size of a *block*, the unit of communication between the user and the server. The bandwidth blowup is $O(\log^2 n)$ and the user space is $O(f_1(n))$ blocks. Our second construction has $n(1+\Theta(\frac{\log n}{B} + \frac{\log\log n}{f_2(n)}))$-bit server space, $O(\log^2 n)$-bandwidth blowup and $O(f_2(n) + R(n))$-user space where $f_2(\cdot)$ is a function such that $f_2(n) = \omega(\log\log n)$ and $O(\log^2 n)$, $R(\cdot)$ is a function with $R(n) = \omega(\log n)$.

For example, if $B = \log^2 n$, $R = \log n \log\log n$, $f_1(n) = f_2(n) = \log n \log\log n$ and $g(n) = \log\log\log n$, the user space is $O(\log n \log\log n)$ in both constructions and the server space is $n(1+\Theta(\frac{\log\log\log n}{\log\log n}))$ (resp. $n(1+\Theta(\frac{1}{\log n}))$) bits in the first (resp. second) construction.

The second construction has better theoretical performance than the first one. However, in practice, with some parameter settings, the first construction also works comparably well as the second construction depending on which performance measure one cares (See Section 5). The first construction is also the basis of the second construction.

If $B = \omega(\log n)$, Goldreich's construction [14] and our constructions are succinct. (Each of these methods works as long as $B \geq c \lg n$ for c around 3.) The assumption $B = \omega(\log n)$ is justified as follows. Stefanov et al. [38] mentioned that the typical block size is 64–256 KB (resp. from 128B to 4KB) in cloud computing scenario (resp. software protection scenario). Even $B \geq \lg^{1.5} n$ holds if $n \leq 2^{6501}$ (resp. $n \leq 2^{97}$) in cloud computing (resp. software protection) scenario with moderate block size of 64KB (resp. 128B).

We achieved exponentially smaller bandwidth blowup compared to Goldreich's construction [14], which is the only preceding non-trivial succinct ORAM construction.

The bandwidth blowup of our constructions are smaller or equal to other non-succinct constructions except [22], [7] and [37]. [22] is based on a very expensive procedure called oblivious sorting and the constant factor of the bandwidth blowup is prohibitively large. [7] has $O(1)$-bandwidth blowup but it requires several assumptions. First, the server needs to perform some computation, e.g., homomorphic encryption evaluation. (In every other

Table 1 Comparison of theoretical performance. Bandwidth blowup is the number of blocks required to be communicated for accessing one block of data. User space includes the temporary space needed during access procedures. n is the database size in bits and B is the block size in bits. B must satisfy $B \geq c_1 \lg n$ and $B = O(n^{c_2})$ for constants $c_1 > 1$, $0 < c_2 < 1$. Typically, c_1 is around 3. $f_1(\cdot)$ is an arbitrary function such that $f_1(n) = \omega(\log n)$ and $O(\log^2 n)$. $f_2(\cdot)$ is an arbitrary function such that $f_2(n) = \omega(\log \log n)$ and $O(\log^2 n)$. $R(\cdot)$ is an arbitrary function such that $R(n) = \omega(\log n)$. $g(\cdot)$ is an arbitrary function such that $g(n) = \omega(1)$ and $o(\sqrt{f_1(n)/\log n})$. Bounds with † are amortized. The method in [7] requires additional assumptions.

	Server space (#bits)	Bandwidth blowup	User space (#block)
Goldreich [14]	$n(1 + \Theta(\frac{\log n}{B} + \frac{1}{\sqrt{n}}))$	$O(\sqrt{n} \log n)^\dagger$	$O(1)$
Kushilevitz, et al. [22]	$n(1 + \Theta(1))$	$O(\frac{\log^2 n}{\log \log n})$	$O(1)$
Stefanov, Shi, Song [37]	$n(1 + \Theta(1))$	$O(\log n)$	$O(n)$
Stefanov et al. [38]	$n(1 + \Theta(1))$	$O(\log^2 n)$	$O(R(n))$
Devadas et al. [7]	$n(1 + \Theta(1))$	$O(1)$	$O(1)$
Our result (Theorem 3)	$n(1 + \Theta(\frac{\log n}{B} + \frac{g(n)}{f_1(n)/\log n}))$	$O(\log^2 n)$	$O(f_1(n))$
Our result (Theorem 5)	$n(1 + \Theta(\frac{\log n}{B} + \frac{\log \log n}{f_2(n)}))$	$O(\log^2 n)$	$O(f_2(n) + R(n))$

construction in Table 1, the server suffices to respond to read/write requests.) [7] also requires a computational assumption (decisional composite residuosity or learning with errors assumption), and larger block size ($B = \widetilde{\omega}(\log^2 n)$ to $\widetilde{\omega}(\log^6 n)$ depending on the case, where $\widetilde{\omega}(\cdot)$ hides a polyloglog factor). [37] takes cn-bit user space where $c \ll 1$. This method is effective for ordinary cloud computing setting but the user space is too large for secure processor setting — the PHANTOM-like applications where server space efficiency is more important.

Possible applications. There are several ORAM application scenarios with different requirements. Our methods are particularly relevant to *secure processor* scenario. In this scenario, it is assumed that a special processor under the control of the user is available in a remote server and the adversary cannot observe the activities inside the processor. The cloud service user sends a piece of code to the trusted processor, which, in turn, executes the code on the server. The communication between the cloud service user and the secure processor is protected by private key encryption. ORAM is implemented inside of the trusted processor using FPGA and it hides the processor's access pattern to the main memory on the server. After executing the code, the secure processor may return the (encrypted) output to the cloud service user. One of the main advantages of this approach over the conventional ORAM application, in which the cloud service user locally executes ORAM, is that ORAM bandwidth blowup applies to the relatively cheap processor–memory communication rather than the costly over-network communication. Note that, with the ORAM user-server terminology, the secure processor (resp. the main memory) is the user (resp. the server).

In secure processor scenario,

- the user space is very limited, e.g., 6MB;
- The server usually does not perform complex computation;
- Simple ORAM algorithms are desirable for hardware implementation;
- The server space is much larger than the user space but there is some noticeable limit. The server can use disks if needed but it greatly slows down accesses.

In most existing secure processor systems, the Path ORAM [38] or its close variants are used [11, 23, 33, 12]. Indeed, the Path ORAM satisfies the first three requirements above.

However, it does not capture the last one. For example, suppose 128GB database is stored in the Path ORAM. If the block size is 128B, it takes about 10G blocks, i.e., 1.28TB (to ensure rigorous security). Then, each ORAM access procedure takes about 31μs assuming each memory access takes 100ns. If half of the 10G blocks are stored in the main memory and the other half is stored in the disk, due to the randomized access pattern of the Path ORAM, almost every ORAM access procedure ends up a disk seek, which takes milliseconds order time. In such cases, it is reasonable to use another ORAM construction that takes, say, half the space of the Path ORAM even though it requires twice as many memory accesses.

Tree-based ORAM. Our ORAM constructions are tree-based. In a typical tree-based ORAM construction, N blocks are stored in a complete binary tree with N leaves on the server. Each node of the tree can store up to Z blocks where Z is a constant. Each block is assigned a position label, a uniformly random integer in $[N]$. A block with position label i must be stored at some node on the path from the root to the i-th leaf. This framework was introduced by Shi et al. [36] and used in many subsequent studies [38, 13, 33, 5, 7].

Consider a particular block b. As the user continuously issues access requests, b moves around the tree in roughly the following manner. First, when the user issues an access request to b, b is picked out of the tree and given a new uniformly random position label. Then, b is inserted into the tree from the root. If the user issues an access request to another block, then, with some probability, b will move down the path to the leaf indicated by its position label. If the next node on the path is full, b must wait for the blocks “ahead” to move down. If the pace at which the blocks move down the tree cannot keep up with the pace at which blocks are picked out and reinserted from the root, then, some blocks will not be able to reenter the tree. If such “congestion” occurs, the user must maintain the overflown blocks locally.

Note that most space in the tree is wasted: there are $2N-1$ nodes in the tree, each with capacity Z, whereas there are only N blocks. Thus, to save server space, it is desirable to make the tree more compact, for example, by reducing Z. However, to maintain a low probability of “congestion”, it is desirable to make the tree larger, for example, by increasing Z. To construct a succinct tree-based ORAM, we need to satisfy these conflicting demands.

Our ideas. One of our key ideas is the following two-stage tree layout. We first change the tree to a complete binary tree with $N/\lg^{1.4} N$ leaves (assume this is a power of 2). In addition, we set the capacity of each leaf node to $\lg^{1.4} N + \lg^{1.3} N$ while keeping the capacity of each internal node at Z. The total size of the leaf nodes is then $N + N/\lg^{0.1} N$, and the total size of all tree nodes except the leaves is $\Theta(N/\lg^{1.4} N)$. Thus, the total size of the entire tree is $N + o(N)$. We choose each position label from $[N/\lg^{1.4} N]$.

To see why blocks can flow around in this tree without much congestion, suppose that the user inserts each block directly into the leaf node pointed to by the block’s position label. Clearly, the loads of leaves in this hypothetical setting dominates the loads of leaves in the real setting. Then, the situation would exactly be the same as the “balls-into-bins” game [24] with N balls and $N/\lg^{1.4} N$ bins. In particular, the number of blocks stored in each leaf node is $\log^{1.4} N + \Theta(\log^{1.2} N)$ with high probability. Thus, every leaf node has sufficient capacity to store all of its assigned blocks. Furthermore, the blocks in the internal nodes flow as smoothly as in the original non-succinct ORAM construction since we did not modify that part. Thus, the blocks flow without much congestion in the tree.

Another key idea follows naturally from the above argument, specifically from the connection to the balls-into-bins game. A remarkable phenomenon known as “the power of two choices" states that, in the balls-into-bins game, if one chooses two bins uniformly and independently for each ball, and throws the ball into the least loaded bin, the bin loads will be distributed much more tightly around the mean than they are in the one-choice game [1, 3, 24]. The maximum bin load corresponds to the leaf node size in tree-based ORAM constructions. Thus, the size of the tree can be further decreased by using the two-choice strategy to assign the position labels. This is the idea behind the construction in Section 4.

We note that the current paper is the first to apply the power of two choices to tree-based ORAM. (Some non-tree-based constructions [30, 16, 22] use the two choices idea in the form of cuckoo hashing [29].) Moreover, the resulting algorithms keep the simplicity of the Path ORAM [38], which is a highly valuable asset in the relevant application scenario as mentioned above. As for the analysis, the existing stash size analyses [38, 33] do not seem to work with parameter regimes required for succinctness. We will give a different proof route (though it still heavily borrows from [38, 33]) in the full version of the paper.

Our contributions. Our contributions in the current paper are as follows:

- We introduce the notion of succinct oblivious RAM. This is a promising first step to systematically design ORAM constructions with small server space usage;
- We propose two succinct ORAM constructions. Not only being succinct, these constructions exhibit state-of-the-art performance in terms of the bandwidth blowup. The methods are simple and easy to implement.
- We also give non-asymptotic bounds and simulation results which indicate that the proposed methods are practically effective.

Related work. In the field of succinct data structures [20, 19], the goal is to represent an object such as a string [26, 34, 17, 9, 15, 35, 21, 10, 18, 27] or a tree [6, 25, 31, 2, 8, 28] in such a way that a) only $OPT + o(OPT)$ bits are required, and b) relevant queries such as random access or substring search are efficiently supported. Here, OPT is the information theoretic optimum, i.e., the minimum number of bits needed to represent the object.

The current study is related to succinct data structures in the following way. Suppose a remote server hosts a database that is implemented by a succinct data structure, and a user wishes to access the database without revealing the access pattern to the server. The user, of course, can apply any existing ORAM constructions. However, if ORAM increases the database size by some constant factor, it destroys the $OPT + o(OPT)$ bound guaranteed by the succinct data structure. One can apply the succinct ORAM constructions proposed in this paper to hide succinct data structure access pattern on a remote storage device without harming the theoretical guarantee on the data structure size.

Notations. We denote the set $\{0, 1, \ldots n-1\}$ as $[n]$ for a non-negative integer n. We write $\lg x$ to denote the base-2 logarithm of x and $\ln x$ to denote the natural logarithm of x. We write $\log x$ to denote the logarithm of x in the context where the base can be any positive constant. We write $\mathrm{poly}(n)$ to denote n^c for some constant $c > 0$. A negligible function of n is defined to be a function that is asymptotically smaller than $1/n^c$ for any constant $c > 0$.

2 ORAM: Preliminaries

Definition. Suppose there are three parties the *user*, the *server* and the *oblivious RAM (ORAM) simulator*. Let each of B and n be a positive integer and $N := n/B$. (We assume n is a multiple of B for brevity.) The value B models the unit of communication and n

models the database size. We call a chunk of B bits a *block*. A *logical (resp. physical) access request* is a triplet $(\mathrm{op}, \mathrm{addr}, \mathrm{val})$, where $\mathrm{op} \in \{\mathrm{read}, \mathrm{write}\}$, $\mathrm{addr} \in [N]$ (resp. $\mathrm{addr} \in \mathbb{N}$), $\mathrm{val} \in \{0,1\}^B$. The user sends logical access requests to the ORAM simulator and receives a block for each request. The server receives physical access requests from the ORAM simulator and returns a block for each request in the following way: for (read, i, v), the server returns v of the most recent request (write, i, v). The ORAM simulator takes a sequence of logical access requests from the user and for each logical access request, it makes a sequence of physical access requests to the server receiving a returned block for each of them, and returns a block to the user. The ORAM simulator is possibly stateful and probabilistic. It must respond to logical access requests online and must satisfy the following conditions:

Correctness The ORAM simulator is correct iff, for a logical access request with $\mathrm{addr} = i$, it returns v of the previous and most recent logical access request (write, i, v);[1]

Security The ORAM simulator is computationally (resp. information theoretically) secure iff, for any logical access request sequences of the same length, the distributions of the addr values of the resulting physical access requests are computationally (resp. information theoretically) indistinguishable.

An ORAM construction is an ORAM simulator implementation. We have distinguished the user from the ORAM simulator for exposition but in practice, an ORAM simulator is a program run by the user. Thus, we do not distinguish them in the rest of the paper.

Encryption. In the ORAM constructions considered in this paper, the user holds a symmetric cipher key and every block is encrypted when it is stored on the server. Though encryption increases the database size, the increase is minor[2]and we ignore the space blowup due to encryption in the rest of the paper.

Performance measures. The most popular ORAM performance measures include the space required by the user/server and time required for each logical access.

In most ORAM constructions, the user needs to maintain a small amount of information locally. In addition to this, in some constructions, the user temporarily need to store more information during the access procedure. We refer the amount of the space the user temporarily needs during access procedure as *temporary space usage* and the amount of the space the user needs even if no access is made as *permanent space usage*.

In this paper, we pay special attention to the server space usage. In particular, we use the following notion of *succinctness* as a criterion for ORAM server-space efficiency:

▶ **Definition 1.** If the server space usage of an ORAM construction representing an n-bit database is $n + o(n)$ bits, the ORAM construction is said to be succinct.

As for the access efficiency, following the previous studies, we use the amount of communication between the user and the server as a proxy for the access time. We define the *bandwidth blowup* of an ORAM construction to be the number of blocks that needs to be communicated between the user and the server per logical access. In other words, the bandwidth blowup is the ratio of communication amount needed for secure access to communication amount needed for ordinary (insecure) access.

[1] We use the convention that not only read but also write requests have return values.

[2] In theory, we can guarantee the semantic security and succinctness at the same time with extra bits of amount $\omega(\log n)$ and $o(B)$ per each block. In practice, assuming that we use "counter mode" block cipher with 128 bits counters and the typical block sizes mentioned in Section 1, the space blowup is 1/4096–1/16384 (resp. 1/8–1/256) factor in cloud computing (resp. software protection) scenario.

Asymptotic behavior of parameters. Among the ORAM-related parameters, the original database size n and block size B are outside of the user's control. Other parameters, e.g., the metadata size, can be chosen by the user. We assume that B is a function of n satisfying $B = \omega(\log n)$. (See Section 1 for the justification.) Thus, after all, n is the only free parameter on which the other parameters depend. In all asymptotic statements in this paper, the limit is taken as $n \to \infty$.

Sub-ORAM. We use an ORAM construction encapsulated into the following proposition as a blackbox. Concretely, the Path ORAM [38] suffices.

▶ **Proposition 2.** *Let n be the database size and B be the block size, in bits. If $B \geq 3 \lg n$ and $B = O(n^c)$ for some $0 < c < 1$, there exists an information theoretically secure ORAM construction such that i) the server's space usage is $n(10 + \Theta(\frac{\log n}{B}))$ bits; ii) the worst-case bandwidth blowup is $O(\log^2 n)$; iii) the user's temporary space usage is $O(\log n)$ blocks; and iv) for any $R = \omega(\log n)$, the probability that the user's permanent space usage becomes larger than R blocks during $\mathrm{poly}(n)$ logical accesses is negligible.*

3 Succinct ORAM Construction

In this section, we prove the following theorem.

▶ **Theorem 3.** *Let n be the database size and B be the block size, both in bits. If $B \geq 3 \lg n$ and $B = O(n^c)$ for some constant $0 < c < 1$, then for any $f : \mathbb{N} \to \mathbb{R}$ such that $f(n) = \omega(\log n)$ and $f(n) = O(\log^2 n)$ and any $g : \mathbb{N} \to \mathbb{R}$ such that $g(n) = \omega(1)$ and $g(n) = o(\sqrt{f(n)/\log n})$, there exists an information theoretically secure ORAM construction such that i) the server's space usage is bounded by $n(1 + \Theta(\frac{\log n}{B} + \frac{g(n)}{\sqrt{f(n)/\log n}}))$ bits; ii) the worst case bandwidth blowup is $O(\log^2 n)$; iii) the user's temporary space usage is $O(f(n))$ blocks; and iv) for any $R = \omega(\log n)$, the probability that the user's permanent space usage becomes larger than R blocks during $\mathrm{poly}(n)$ logical accesses is negligible.*

▶ **Corollary 4.** *If $B = \omega(\log n)$, then, the ORAM construction of Theorem 3 is succinct.*

3.1 Description

For the clarity of explanation, we first describe a simplified ORAM construction where the user needs to maintain a large amount of information locally. Then, we obtain an ORAM construction with the claimed bounds by slightly modifying the simplified construction.

As we mentioned in Section 1, in a tree-based ORAM construction, blocks on the server are stored in the nodes of a complete binary tree. The key point of the method in this section is the choice of the tree height L and the leaf node capacity M. Specifically, in the rest of this section, let $L := \lceil \lg \frac{N}{f(n)} \rceil$ and $M := \lceil \frac{N}{2^L} + g(n)\sqrt{\frac{NL}{2^L}} \rceil$ where $N := n/B$. We assume, for brevity, that each of $\lg \frac{N}{f(n)}$ and $\frac{N}{2^L} + g(n)\sqrt{\frac{NL}{2^L}}$ is an integer.

Block usage. The ORAM is supposed to provide the user with an interface to access the database as if it is stored in array A of B-bit blocks (Section 2). We use blocks as follows :

- Each block is either a *data block* or a *metadata block*;
- Each data block is either a *real block* or a *dummy block*. A real block contains an entry of A. A dummy block does not contain any information on the database contents and is used only to hide the access pattern;

- Each real block is given a *position label*, a value in $[2^L]$;
- A metadata block contains the metadata of several data blocks. For each data block, its metadata consists of
  - type: A flag indicating whether the block is real or dummy;
  - addr: If the block is real and represents $A[i]$, the value of addr is i. If the block is a dummy, the value is arbitrary;
  - pos: If the block is real with position label i, the value of pos is i. If the block is a dummy, the value is arbitrary.

Data layout. The server maintains a tree containing data blocks, which we call *data tree*, and another tree containing metadata blocks, which we call *metadata tree*. The data tree is used in such a way that at each point of time, it contains most real blocks with high probability. The user maintains *stash*, which contains the real blocks that are not in the data tree, and *position table*, which contains the position labels of all real blocks.

The data tree is a complete binary tree with 2^L leaves. Each node of the tree is a *bucket*, which is a container that can accommodate a certain number of blocks. We call the buckets corresponding to the internal nodes as *internal buckets* and the buckets corresponding to the leaf nodes as *leaf buckets*. The size of each internal bucket is Z (blocks) while the size of each leaf bucket is M (blocks). We will determine Z to be 3 in the full version of the paper but for now, we consider it as an arbitrary constant. The data tree is represented as the bitstring derived by concatenating all buckets in breadth first order. As is well-known, with this representation, given an index of a node, the index of the parent or left/right child can be derived by simple arithmetic. The total space usage of the data tree is equal to the sum of the bucket sizes.

The metadata tree is also a complete binary tree with 2^L leaves. Each node of the tree is the metadata of the data blocks in the corresponding bucket of the data tree. The metadata tree is represented similarly to the data tree but there is a subtlety. If the metadata of the blocks in a bucket has a size smaller than B, it is wasteful to allocate one full block for them. To avoid this waste, we represent metadata tree as the bitstring derived by concatenating the metadata of all data blocks in the data tree in breadth first order. The space usage of the metadata tree is equal to the sum of all metadata of all data blocks.

Each real block in the stash is maintained with its addr and pos. The stash can be any linear-space data structure that efficiently supports insertion, deletion and range query by pos, e.g., a self balancing binary search tree.

The position table stores the position label of real block storing $A[i]$ in the i-th entry.

Access procedure. Access requests are processed in such a way that the following invariant conditions are always satisfied:

- Each real block is stored either in the data tree or in the stash;
- If a real block with position label ℓ is stored in the data tree, it is in the bucket on the path from the root to the ℓ-th leaf.

Below, we give a high-level description of the main routine and we will provide the pseudocode in the full version of the paper. To read the explanation here, it should suffice to know that $P(\ell)$ means the path from the root to the ℓ-th leaf of the data tree.

Let b_a be the accessed block. We first read the position label ℓ of b_a from the position table and update the position table entry to a number chosen uniformly at random from $[L]$, which will become the new position label of b_a after the access operation is finished. By the invariant conditions above, b_a is either in the stash or $P(\ell)$. We scan $P(\ell)$ and retrieve b_a

if it is in $P(\ell)$. If b_a was not in $P(\ell)$, we retrieve it from the stash. If the current request is a write request, we update the block contents to the new value. Then, we insert b_a with the updated position label and the possibly updated value into the stash. After that, we perform EvictPath operation. The purpose of this operation is a) to move back the blocks in the stash into the tree and b) to move the real blocks in the tree downwards (far from the root). To do this, EvictPath retrieves all real blocks in the path $P(\text{BitReversal(G)})$ (to be explained shortly) into the stash and then, going up $P(\text{BitReversal(G)})$ from leaf to the root, tries to move as many blocks in the stash into the buckets on the path. If some blocks are left in the stash after EvictPath, the user keeps them charging the permanent space usage. Lastly, the value stored at b_a is returned.

The function $\text{BitReversal}(\cdot)$ takes an L-bit integer x and returns the bit reversed version of x while G is the number of Access operations called so far (modulo 2^L). This BitReversal-based scheduling of EvictPath was first proposed by Gentry et al. [13] and is advantageous to keep the stash size small. It also enables to simplify stash size analysis, which we will provide in the full version. Here, it suffices to note that G (and $\text{BitReversal}(G)$) is independent of the accessed database locations.

Outsourcing position table. In the construction described so far, the user space usage is much larger than the bound claimed in Theorem 3 since the user needs to maintain the position table locally. To obtain Theorem 3, we modify the construction so that the position table is stored on the server using the Path ORAM [38]. Accesses to position tables are replaced by a Path ORAM write.

3.2 Analysis

Security. Fix $t > 0$. Let $\mathbf{a}$ be a length $t > 0$ sequence of logical addresses to be accessed and $\mathbf{a}'$ be the corresponding sequence of physical addresses (indices of the server memory) to be accessed. The sequence $\mathbf{a}'$ is determined by $\mathbf{a}$ and the randomness used by the ORAM simulator. To prove the information theoretic security, it suffices to show that $\mathbf{a}'$ really does not depend on $\mathbf{a}$. The sequence $\mathbf{a}'$ consists of $\mathbf{a}'_1$, the physical addresses accessed in the recursive access call to the Path ORAM and $\mathbf{a}'_2$, those accessed in the rest parts. The addresses $\mathbf{a}'_1$ is determined by the Path ORAM access procedure and is independent of $\mathbf{a}$ due to the information theoretic security of the Path ORAM. The addresses $\mathbf{a}'_2$ consists of addresses accessed by $\text{ReadPath}(\ell, a)$ and $\text{EvictPath}()$. $\text{ReadPath}(\ell, a)$ accesses the path $P(\ell)$, which is determined by ℓ, the position label of the accessed block. Since the position labels are chosen independently and uniformly at random, the ReadPath accesses are independent of $\mathbf{a}$. EvictPath accesses $P(\text{BitReversal}(G))$, which is determined by G, the number of times Access was called (modulo 2^L). Thus, the accesses of EvictPath is also independent of $\mathbf{a}$. Therefore, $\mathbf{a}'$ is independent of $\mathbf{a}$.

Server space. First, it is helpful to observe that $\log N = \Theta(\log n)$, $L = \Theta(\log n)$ and $M = \Theta(f(n))$. Remember that the server holds the data tree, the metadata tree and the position table. The total size of the internal (resp. leaf) buckets is $Z(2^L - 1)$ (resp. $M2^L$) blocks. Since $Z(2^L - 1) < Z2^L = ZN/f(n)$ and $M2^L = N + g(n)\sqrt{NL2^L} = N(1 + \Theta(\frac{g(n)}{\sqrt{f(n)/\log n}})) = N(1 + \Theta(h(n)))$ where $h(n) := \frac{g(n)}{\sqrt{f(n)/\log n}}$, the number of the blocks in the data tree is bounded by $ZN/f(n) + N(1 + \Theta(h(n))) = N(1 + \Theta(\frac{1}{f(n)} + h(n)))$.

The metadata for each data block takes 1 bit for type, $\lceil \lg N \rceil$ bits for addr and L bits for pos. The total is $\Theta(\log n)$ bits $= \Theta(\frac{\log n}{B})$ blocks. Thus, the number of bits in the data tree and the metadata tree combined is $BN(1+\Theta(\frac{1}{f(n)}+h(n)))(1+\Theta(\frac{\log n}{B})) = n(1+\Theta(\frac{\log n}{B}+h(n)))$. The position labels take $NL = n\frac{L}{B} \le n\frac{\lg n}{B}$ bits. By Proposition 2, the Path ORAM containing the position table takes $\Theta(n\frac{\log n}{B})$ bits. Thus, the server space is $n(1+\Theta(\frac{\log n}{B}+h(n)))$ bits.

Bandwidth blowup. The bandwidth cost of each of ReadPath and EvictPath is proportional to the sum of the numbers of the blocks in a root–leaf path in the data tree and the metadata tree. The number for the data tree is $ZL + M = O(\log n) + O(f(n)) = O(f(n))$. The number for the metadata tree is around $\frac{2\lg N+1}{B} = o(1)$ factor of that for the data tree. The bandwidth cost for accessing the position table is $O(\log^2 n)$ by Proposition 2. Therefore, the bandwidth blowup of Access is $O(\log^2 n)$.

User space. The temporary user space usage is proportional to the sum of the numbers of the blocks in a root–leaf path in the data tree and the metadata tree. As is shown in the bandwidth analysis, the latter is bounded by $O(f(n))$. We prove the bound on the permanent user space usage, i.e., the stash size in the full version of the paper.

4 Succincter ORAM Construction

In this section, we prove the following theorem.

▶ **Theorem 5.** *Let n be the database size and B be the block size, both in bits. If $B \ge 3\lg n$ and $B = O(n^c)$ for some $0 < c < 1$, then for any $f : \mathbb{N} \to \mathbb{R}$ such that $f(n) = \omega(\log\log n)$ and $f(n) = O(\log^2 n)$, there exists an information theoretically secure ORAM construction for which i) the server's space usage is bounded by $n(1+\Theta(\frac{\log n}{B}+\frac{\log\log n}{f(n)}))$ bits; ii) the worst case bandwidth blowup is $O(\log^2 n)$; iii) the user's temporary space usage is $O(\log n + f(n))$ blocks; and iv) for any $R = \omega(\log n)$, the probability that the user's permanent space usage becomes larger than R blocks during $\mathrm{poly}(n)$ logical accesses is $n^{-\omega(1)}$.*

▶ **Corollary 6.** *If $B = \omega(\log n)$, then, the ORAM construction of Theorem 5 is succinct.*

4.1 Description

As in Section 3, we first explain a simplified version with a large user space usage, and construct the full version that achieves the claimed bounds from the simplified version.

Let $L := \lceil \lg(N/f(n)) \rceil$ and $M := \lceil N/2^L + (1+\varepsilon)\lg L \rceil$ where $N := n/B$ and $\varepsilon > 0$ is a constant. We assume, for brevity, that $\lg(N/f(n))$ and $N/2^L + (1+\varepsilon)\lg L$ are integers.

Block usage. The block usage is the same as the ORAM construction described in Section 3 except that each real block is given *two* position labels instead of one. We call them the *primary position label* and the *secondary position label.* Only the primary position labels are stored in the metadata blocks (as in Section 3).

Data layout. The data layout is basically the same as in Section 3. We only explain the differences from Section 3. First, the position table stores both the primary position labels and the secondary position labels. Second, the user maintains an additional table called *counter table.* It is a size 2^L array whose i-th entry is the number of real blocks with primary position label i. Last, since the value of each of L and M is different from that in Section 3, the tree/bucket size is changed accordingly.

Access procedure. The same invariant conditions as Section 3 are maintained except that the "position label" in the second condition is replaced by "primary position label".

We provide the pseudocode in the full paper. To read the high-level description below, it suffices to know that $P(\ell)$ is the path from the root to the ℓ-th leaf in the data tree.

Let b_a be the accessed block. We first retrieve the two position labels ℓ_1 and ℓ_2 of b_a from the position table and update each of the two position table values to a number chosen independently and uniformly at random from $[L]$, which will become the new position labels of b_a. One of ℓ_1 and ℓ_2 is the primary position label and the other is the secondary position label but we do not know (and do not need to know) which is which. By the invariant conditions, b_a is either in the stash or in $P(\ell_1)$ or $P(\ell_2)$. We scan $P(\ell_1)$ and $P(\ell_2)$ and retrieve b_a from $P(\ell_i)$ if the primary position label is ℓ_i and b_a is in $P(\ell_i)$. If b_a is not found in the paths, it must be in the stash and we retrieve it from the stash. At this point, we know the primary position label ℓ of b_a (since it is written in the pos entry of the block) and we decrement the ℓ-th entry of the counter table, determine the new primary position label ℓ'_i and increment the ℓ'_i-th entry of the counter table. After, that, we update the block contents if it is a write request, call EvictPath and returns the block contents (before update) in the same way as the algorithm in Section 3.

Outsourcing the position/counter table. In the full version of the construction, the position table and the counter table are stored on the server using the Path ORAM. Every read from (resp. write to) each of these tables is done using the Path ORAM access procedure.

4.2 Analysis

Security. The security proof of the current ORAM construction is almost the same as in Section 3. The only difference in the situation is that now, the sequence of accessed addresses $\mathbf{a}'_2$ depends on two position labels instead of one. Anyway, these position labels are distributed independently and uniformly at random and thus, are independent of $\mathbf{a}$.

Server space. The bounds $\log N = \Theta(\log n)$, $L = \Theta(\log n)$ and $M = \Theta(f(n))$ still hold.

The number of blocks in the leaf buckets is $M2^L = N(1 + \frac{(1+\varepsilon)\lg L}{f(n)}) = N(1 + \Theta(\frac{\log\log n}{f(n)}))$. The number of blocks in the internal buckets is $Z(2^L - 1) < ZN/f(n)$, which is $O(\frac{\log\log n}{f(n)})$. Thus, the data tree size is bounded by $N(1+\Theta(\frac{\log\log n}{f(n)}))$ blocks. As in Section 3, the metadata size of each data block is $\Theta(\frac{\log n}{B})$ blocks. Thus, the number of blocks in the data tree and the metadata tree combined is at most $1 + \Theta(\frac{\log n}{B})$ times larger than $N(1 + \Theta(\frac{\log\log n}{f(n)}))$, which is $n(1 + \Theta(\frac{\log n}{B} + \frac{\log\log n}{f(n)}))$ bits.

Position labels take $2NL = 2nL/B \le 2n\frac{\log n}{B}$ bits while counter table values take $2^L\lceil \lg N \rceil = N\lceil \lg N \rceil / f(n) \le N = n/B$ bits. By Proposition 2, the Path ORAM containing the position table (resp. counter table) takes $\Theta(n\frac{\log n}{B})$ (resp. $\Theta(n/B)$) bits.

Therefore, the server space usage is bounded by $n(1 + \Theta(\frac{\log n}{B} + \frac{\log\log n}{f(n)}))$ bits.

Bandwidth blowup. By the same argument as in the bandwidth analysis, the bandwidth cost of each of ReadPath and EvictPath is proportional to $ZL + M = O(\log n + f(n))$ (in blocks). By Proposition 2, the bandwidth cost of access to each of the position table and the counter table is $O(\log^2 n)$. Thus, the bandwidth blowup is $O(\log^2 n)$.

Table 2 Performance comparison with concrete parameters. The symbol † means the integration of Ring ORAM techniques. $N = 2^{20}$, $B = 2^{10}$. A and S are parameters for the Ring ORAM. (A specifies the infrequency of EvictPath and S is the space in each bucket reserved for dummy blocks.) The cost for recursive calls and metadata handling are relatively minor and not included. The stash overflow probability is $< 2^{-80}$ for rigorous settings. Aggressive settings do not have security guarantees (stash size bounds) and, in particular, are not suitable for fair comparison.

		Parameters Z, L, M, A, S	Extra server space	Bandwidth	Stash size
Rigorous	[38]	5,20,–,–,–	$9N$	210	114
	[32]	5,19,–,4,6	$10N$	109	63
	Th. 3	3,15,112,–,–	$2.59N$	471	32
	Th. 3†	5,15,112,4,7	$2.91N$	253	64
Aggressive	[38]	4,19,–,–,–	$3N$	160	
	[32]	5,19,–,4,6	$7N$	145	
	Th. 3	4,15,36,–,–	$.25N$	288	
	Th. 3†	5,15,36,4,6	$.46875N$	163	
	Th. 5	3,16,14,–,–	$.0625N$	248	
	Th. 5†	5,15,28,4,7	$.25N$	194	

User space. By the same argument as in the user space analysis in Section 3, the temporary user space is proportional to $ZL + M = O(\log n + f(n))$. We prove the bound on the permanent user space usage, i.e., the

5 Practicality of the Proposed Methods

Table 2 shows the performance of the proposed methods, the Path ORAM [38] and the Ring ORAM [32] with concrete parameters. The Ring ORAM has asymptotically the same performance as the Path ORAM but it achieves constant factor smaller bandwidth at the cost of larger server space. It is easy to integrate the main technique of the Ring ORAM to the internal nodes of the proposed methods and we also show the performance of these variants. We show the integration itself in the full paper.

The table contains "rigorous" and "aggressive" parameter settings. Rigorous parameters were derived from theoretical analysis with additional care for constant factors. The aggressive parameters for existing methods were taken from the experiments in the original papers. We chose the aggressive parameters for the proposed methods by simulation: we simulated database scan (accessing addresses $1, 2, \dots, N$) for 100 times and found some parameters for which the stash size after every scan was zero. (Such usage of scan is standard in literature since scan maximizes the stash size.) We emphasize that constructions with aggressive parameters lack rigorous security and they are not suitable for fair comparison.

Unfortunately, we could not derive rigorous bounds for the second construction (Theorem 5) for reasonable size of N since the balls-into-bins analysis of Berenbrink et al. [3], used in the stash size analysis, requires a very large number of bins. However, the simulation results indicate that the second construction works for reasonable size of N.

6 Conclusion

ORAM is a multifaceted problem and recently, researchers have been recognizing the importance of rethinking the relevancy of multiple aspects of ORAM using modern standards [37, 4].

In this paper, we provided another point of view and insight for this exploration by introducing the notion of succinctness to ORAM and proposing succinct ORAM constructions. We think our methods are particularly suitable for secure processor setting. It is interesting to consider succinct constructions optimized for other settings.

References

1 Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999. `doi:10.1137/S0097539795288490`.

2 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. `doi:10.1007/s00453-004-1146-6`.

3 Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: the heavily loaded case. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 745–754. ACM, 2000. `doi:10.1145/335305.335411`.

4 Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 837–849. ACM, 2015. `doi:10.1145/2810103.2813649`.

5 Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with õ(log^2 n) overhead. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 62–81. Springer, 2014. `doi:10.1007/978-3-662-45608-8_4`.

6 David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, pages 383–391, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=313852.314087`.

7 Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, volume 9563 of *Lecture Notes in Computer Science*, pages 145–174. Springer, 2016. `doi:10.1007/978-3-662-49099-0_6`.

8 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 184–196. IEEE Computer Society, 2005. `doi:10.1109/SFCS.2005.69`.

9 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. `doi:10.1145/1082036.1082039`.

10 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007. `doi:10.1145/1240233.1240243`.

11 Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the 7th ACM Workshop on Scalable Trusted Computing*, STC '12, pages 3–8, New York, NY, USA, 2012. ACM. `doi:10.1145/2382536.2382540`.

12 Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 103–116. ACM, 2015. doi:10.1145/2694344.2694353.

13 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 Emiliano De Cristofaro and Matthew K. Wright, editors, *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013. doi:10.1007/978-3-642-39077-7_1.

14 Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194. ACM, 1987. doi:10.1145/28395.28416.

15 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 368–373, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1109557.1109599.

16 Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'11, pages 576–587, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2027223.2027282.

17 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

18 Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *J. ACM*, 61(2):9:1–9:36, 2014. doi:10.1145/2590774.

19 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

20 Guy Joseph Jacobson. *Succinct Static Data Structures.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1988.

21 Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 575–584, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1283383.1283445.

22 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 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156. SIAM, 2012. doi:10.1137/1.9781611973099.13.

23 Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM*

SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013, pages 311–324. ACM, 2013. doi:10.1145/2508859.2516692.

24 Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis.* Cambridge University Press, New York, NY, USA, 2nd edition, 2017.

25 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

26 J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001. doi:10.1006/jagm.2000.1151.

27 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014. doi:10.1137/130908245.

28 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. doi:10.1145/2601073.

29 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. doi:10.1016/j.jalgor.2003.12.002.

30 Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 502–519, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1881412.1881447.

31 Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 357–368. Springer, 2003. doi:10.1007/3-540-45061-0_30.

32 Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious ram. In *24th USENIX Security Symposium*, pages 415–430, Washington, D.C., 2015. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-ling.

33 Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious ram in secure processors. *SIGARCH Comput. Archit. News*, 41(3):571–582, 2013. doi:10.1145/2508148.2485971.

34 Kunihiko Sadakane. Succinct representations of *Lcp* information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 225–232, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=545381.545410.

35 Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 1230–1239, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1109557.1109693.

36 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 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2011. doi:10.1007/978-3-642-25385-0_11.

37 Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious ram. In *19th Annual Network and Distributed System Security Symposium*, 2012. URL: http://www.internetsociety.org/towards-practical-oblivious-ram.

38 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 Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310. ACM, 2013. `doi:10.1145/2508859.2516660`.