Oblivious RAM

Classical results and recent developments

Marios Georgiou

October 21, 2016

Second Exam

Committee

Prof. Nelly Fazio – CCNY (Mentor)
Prof. Rosario Gennaro – CCNY
Prof. William E. Skeith III – CCNY

Computer Science Department
The Graduate Center
City University of New York
Abstract

This survey presents classical results as well as recent developments in the cryptographic area of Oblivious RAM (ORAM). An ORAM, first studied by Goldreich and Ostrovsky [Gol87, Ost90, GO96] is an access protocol between two parties, the CPU and the memory, with the following security property: the data stored in the memory as well as the access pattern to these data remains hidden from the memory. We explore different aspects that have to do with the communication and space complexity of the CPU and the memory, the round complexity per access and the parallel complexity in the case where multiple CPUs share a common memory. Moreover, we study how these parameters are affected when the RAM is actually a server capable of computation. In particular, this setting circumvents the $\Omega(\log n)$ lower bound [GO96] on the communication overhead given $O(1)$ local storage, thus enabling constructions with constant overhead.
List of Figures

1. Hierarchical ORAM example. Here we have $k = 2$ and so the first buffer has $2^2$ buckets and $L = 5$ so the last buffer has $2^5$ buckets. The size of each bucket is $b = 4$.

2. Cuckoo hashing example. Suppose that we have the values: $h_1(i) = 10, h_2(i) = 5, h_1(j) = 13, h_2(j) = 5, h_1(k) = 16, h_2(k) = 13$. Suppose also that the initial state of the memory is the one appearing in figure 2a and that we want to insert the block $M[l]$ such that $h_1(l) = 10$. The first figure shows how the existing blocks will be kicked out in order to allow the new block reside in address 10; the arrows show where the next address of the block will be. Figure 2b shows the state of the memory at the end of the insertion.

## List of Tables

1. An non-exhaustive list of works created as concatenation of tables from [SVDS+13, SCSL11, PR10, WCS15]. Comparison of different classical ORAM schemes in terms of minimum block size, client storage and worst case communication overhead. Here, $n$ denotes the total number of blocks. [CLP14] achieves statistical security. Schemes with overhead more than $n$, are mostly famous for their amortized cost. . . 11

2. A non-exhaustive list of results achieving constant communication overhead assuming server computation. The table is a concatenation of tables from [MMB15, DvDF+16]. Here, $B$ is the block size, $\lambda$ is the security parameter and $n$ is the total number of blocks. . . . . . . . . . . . . . . . . . . . . . . . 13

3. A comparison between the two available (to the best of our knowledge) schemes for oblivious PRAM that achieve only a polylogarithmic slowdown. . . . . . . . . . . . . . . . . . . . . . . . 15

4. A non exhaustive list of constructions with only two rounds of interaction per access copied partly from [GMP15]. The last column corresponds to the worst case overhead. Here again, $n$ is the total number of blocks. . . . . . . . . . . . . . . . . . . . . . . . 25
1 Introduction

An oblivious RAM construction aims to address the following problem. Suppose that there is a client/CPU that wants to perform computations on data that is stored on an untrusted server/RAM. The client does not trust the server in the sense that it does not want the server to know anything about the computation; i.e. neither the data stored in the server nor the pattern by which the client accesses these data. For example, we would not like the server to know if we accessed the same element (block) twice. It is therefore trivial to see that just encrypting the blocks using a semantically secure scheme is not enough since reading the same block twice is obvious to the server. In some sense ORAM aims to hide this information by regularly shuffling the data so that a block is never read twice from the same location. The study of ORAM was initiated in the seminal paper by Golreich and Ostrovsky [Gol87, Ost90, GO96].

A trivial way to achieve ORAM is the following. The client who wants to store $n$ blocks, sends to the server the blocks encrypted. When the client wants to access a block $i \in [n]$, it queries one by one all the blocks, re-encrypts them and sends them back to the server. Only when it queries the $i$th block it keeps it and works with it. Notice that in this case the server cannot distinguish between any two different access patterns since every read or write on the memory has exactly the same behavior. The drawback of this construction is its efficiency. In particular, each actual read/write operation incurs an overhead of $\Theta(n)$ virtual reads/writes. All subsequent works on ORAM try to tackle such efficiency parameters; i.e. the client and the server space, the communication overhead as well as the number of interactions.

On the other hand, we cannot hope to achieve extreme efficiency. In
particular, [GO96] shows that if we desire constant client space and $O(n)$ server space, then there is a lower bound of $\Omega(\log n)$ communication overhead (Section 4). A natural question that comes then is whether this lower bound is tight. Wang et al. [WCSI15] have managed to achieve a scheme with optimal circuit and communication overhead assuming that the block size is at least $\Omega(\log^2 n)$.

On a different line of research and on a race to overcome the lower bound, the community started constructing more sophisticated schemes where the server has the ability to perform computations [GGH+13, AKST14, MMB15, DvDF+16]. Sometimes this is called Verifiable Oblivious Storage or VOS to differentiate from classic ORAM. Indeed the lower bound of [GO96] does not hold anymore if the server is allowed to perform computations. In the state of the art, [MMB15] provided schemes with $O(1)$ communication overhead and polylog($n$) server computation overhead where the block size is $\Omega(\log^4 n)$ (in contrast to $\Omega(\log^6 n)$ in [DvDF+16]). [AKST14] provides a scheme achieving similar parameters but is based on Fully Homomorphic Encryption (FHE). Last, [GGH+13] achieves security only in the semi-honest model.

In TCC2016a, two papers appeared [BCP16, CLT16] that deal with the case of parallel ORAMs. [BCP16] presents a compiler from any ORAM to a OPRAM with only a polylogarithmic slowdown in total and parallel complexity and communication overhead $\omega(\log^3 n)$ and [CLT16] drops this communication overhead down to $O(\log^2 n)$.

In terms of round complexity, Garg et al. [GMP15] proposed a construction with exactly two rounds of communication using garbled circuits.
2 Definitions

Most constructions of oblivious RAM fall into two main categories. Chronologically first are the hierarchical ORAMs, where the memory is modeled as a sequence of buffers whose size follows a geometric series. It was introduced in [GO96] and improved by various works including the important works of Goodrich et al. [GMOT12] that gives a construction with $O(\log^2 n)$ overhead and Kushilevitz et al. [KLO12] with $O(\log^2 n / \log \log n)$ overhead. The second category is the tree-based ORAMs introduced by Shi et al. [SCSL11] and later improved at [SVDS+13] where the memory is modeled as a tree and blocks reside in the nodes of this tree.

In this survey, we will be interested in the following complexity measures. Client and server storage, communication overhead per operation, where in classic ORAMs cannot fall under $\Omega(\log n)$ and in VOS (ORAM where the server can perform computations to improve efficiency) constant communication overhead can be achieved. Block size is also important in some cases since the overhead is computed as the communication complexity over the block size. Therefore big blocks can actually affect the asymptotics and achieve better results. For example, if a scheme achieves communication complexity of $O((B + \log^3 n) \cdot \log n)$, where the block size $B = \log^3 n$ then the overhead is only $O(\log n)$ whereas if $B = \Omega(\log n)$ the overhead is $O(\log^3 n)$. Last, we are interested in round complexity, parallel complexity in case of OPRAMs and computation overhead in ORAM with server computation.

2.1 ORAM

Intuitively, an ORAM is a protocol between a CPU/client and a RAM/server with the property that the server cannot distinguish the access patterns of
two equal-size query sequences. This can be formalized using a standard indistinguishability definition or a simulation-based definition.

**Definition 2.1 (ORAM [GMP15])**. An ORAM is a pair of interactive protocols (Setup, Access) with the following properties:

- \((s, EM) \leftarrow \text{Setup}(M, \perp)\) takes as input a memory \(M\) from the client and returns an encrypted memory \(EM\) to the server and a secret state \(s\) to the client.

- \(((M[i], s'), EM') \leftarrow \text{Access}((s, i, v), EM)\) takes as input from the client a secret state \(s\), a memory identifier \(i\) and possibly a value \(v\), from the server an encrypted memory \(EM\) and returns to the client the memory block \(M[i]\) and to the server an encryption \(EM'\) of an updated memory \(M'\) so that \(M'[i] = v\), if \(v \neq \perp\).

**Definition 2.2 (Correctness)**. The ORAM is correct if for any adversary \(A\) who returns memory \(M_0\) and a sequence of accesses \((i_1, v_1), \ldots, (i_q, v_q)\), and for any \(j \in [q]\), it holds that \(M_j[i_j] = \text{OrMem}_j[i_j]\) where \(\text{OrMem}\) is the original memory that would occur if we ran a regular read/write protocol and \(((M_j[i_j], s_j), EM_j) = \text{Access}((s_{j-1}, i_j, v_j), EM_{j-1})\).

**Definition 2.3 (Security)**. Let \(\vec{y} = (i_1, v_1), \ldots, (i_q, v_q)\) be a sequence of accesses to the ORAM and let \(A(\vec{y})\) be the access pattern of \(\vec{y}\). The ORAM is secure (oblivious) if for any \(\vec{y}_1, \vec{y}_2\) such that \(|\vec{y}_1| = |\vec{y}_2|\) it holds that \(A(\vec{y}_1) \sim_c A(\vec{y}_2)\).

Although the game based definition above captures the properties that we expect from an ORAM, Garg et al. [GMP15] have proposed a neat and possibly stronger simulation based definition.
Definition 2.4 (Simulation based \textsuperscript{[GMP15]} – SemiHonest). The ORAM is secure if for any PPT adversary $A$ there is a simulator $\text{Sim}$ such that the following experiments are indistinguishable.

- **Real$_A$:** A outputs a memory $M$ and queries $(i_1,v_1),\ldots,(i_q,v_q)$, where $v_j = \bot$ if it is a read operation. Then the experiment runs $\langle s_0, EM_0 \rangle \leftarrow \text{Setup}(M, \bot)$ and $\langle (M[i_j], s_j), EM_j \rangle \leftarrow \text{Access}(\langle s_{j-1}, i_j, v_j \rangle, EM_{j-1})$. Finally the experiment outputs the transcript of the protocol.

- **Ideal$_A,\text{Sim}$:** The experiment outputs $\text{Sim}(q, |M|)$.

It is worth to note that several similar definitions of ORAM have been given. An interesting alternative has been given by Shi et al. \textsuperscript{[SCSL11]} where the ORAM is defined by operations ReadAndRemove and Add, with the obvious meaning, instead of using Access to either read or write. This is because in all non-trivial implementations, a write operation is always preceded by a dummy read and remove operation, so in some sense these are the fundamental operations.

3 Overview of results and comparisons

In the following we mention the important results in the area giving brief comments where necessary and elaborating on the important aspects.

3.1 Classical ORAM

For Classical ORAM (where the server does not have computational power), a list of different results appears in table \textsuperscript{1}. The first scheme was proposed by Goldreich and Ostrovsky \textsuperscript{[GO96]} which achieved $O(n \log^2 n)$ worst case communication overhead but only $O(\log^3 n)$ amortized overhead. This was
<table>
<thead>
<tr>
<th>Scheme</th>
<th>Block Size</th>
<th>Client Storage</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVDS$^{+13}$</td>
<td>$\Omega(\log n)$</td>
<td>$O(\log n) \cdot \omega(1)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td>SVDS$^{+13}$</td>
<td>$\Omega(\log^2 n)$</td>
<td>$O(\log n) \cdot \omega(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>CLP14</td>
<td>$\Omega(\log n)$</td>
<td>$O(\log^{2+\epsilon} n)$</td>
<td>$O(\log^2 n \cdot \log \log n) \cdot \omega(1)$</td>
</tr>
<tr>
<td>GGH$^{+13}$</td>
<td>$\Omega(\log n)$</td>
<td>$O(\log^{2n} n) \cdot \omega(1)$</td>
<td>$O(\log^3 n / \log \log n) \cdot \omega(1)$</td>
</tr>
<tr>
<td>GO96</td>
<td>$\Omega(\log n)$</td>
<td>$O(1)$</td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td>PR10</td>
<td>$\Omega(\log n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>SCSL11</td>
<td>$\Omega(\log n)$</td>
<td>$O(1)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td>GMOT12</td>
<td>$\Omega(\log n)$</td>
<td>$O(1)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td>KLO12</td>
<td>$\Omega(\log n)$</td>
<td>$O(1)$</td>
<td>$O(\log^2 n / \log \log n)$</td>
</tr>
<tr>
<td>WCS15</td>
<td>$\Omega(\log^2 n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Table 1: An non-exhaustive list of works created as concatenation of tables from SVDS$^{+13}$, SCSL11, PR10, WCS15. Comparison of different classical ORAM schemes in terms of minimum block size, client storage and worst case communication overhead. Here, $n$ denotes the total number of blocks. CLP14 achieves statistical security. Schemes with overhead more than $n$, are mostly famous for their amortized cost.

the first paper that introduced the idea of modeling the RAM as a hierarchical data-structure where the ORAM consists of buckets whose size increases in a geometrical series and buckets are first stored in the top smallest bucket and gradually shift down towards bigger and bigger buckets (see section 5.1 for details). GMOT12, KLO12 are also based on the hierarchical data-structure. Interestingly, KLO12 gives several cryptanalyses for previous schemes that use cuckoo hashing (see section 5.1.1) incorrectly.

An important breakthrough was the result of Shi et al. SCSL11 who presented a totally different memory structure based on trees. In particular, in tree-based ORAMs, the memory is modeled as a tree and a block is associated with a path and resides somewhere in this path. Now, reading the block is accomplished by reading the entire path associated with it, while a block is always written in the root. (see section 5.2 for details). An important subprocess in such a structure has to do with evicting blocks from upper to lower nodes in their corresponding path. Eviction can be
a very costly operation and optimization of it can improve a lot the efficiency of the algorithm (see section 5.2.1 for more details). In the second part of table 1 we can see a line of results that achieve better and better communication complexity while preserving the constant client storage. Importantly, [WCSI15] presents a construction of high efficiency in terms of circuit complexity of the client (and thus communication overhead), which achieves optimal overhead and thus proves that the lower bound of Goldreich and Ostrovsky is tight for blocks of size $\Omega(\log^2 n)$. The way to achieve this, is to perform a very efficient eviction procedure that involves scanning the path 3 times and keeping metadata before actually applying the eviction.

All constructions in table 1 are based on the tree structure except the seminal work of Goldreich and Ostrovsky [GO96] the one of Goodrich et al. [GMOT12] and Kushilevitz et al. [KLO12]. [CLP14] achieves better overhead than [SCSL11] by modifying the bucket size, using a stash and following a more efficient eviction procedure.

### 3.2 RAM with server computation (VOS)

As we mentioned in the introduction, allowing the server to perform computation can overcome the Goldreich-Ostrovsky lower bound [GO96] (see also section 4) and indeed there are scheme that achieve constant overhead [AKST14, MMB15, DvDF+16]. Table 2 presents a comparison between different results. Although previous results already existed, [AKST14] was the first paper that made explicit the distinction between regular ORAM and ORAM with server computation (called verifiable oblivious storage or VOS there). In their construction, they present a universal compiler that takes any tree-based ORAM scheme and using FHE, drops the communication overhead. The idea was to use FHE in order to avoid communicating
Table 2: A non-exhaustive list of results achieving constant communication overhead assuming server computation. The table is a concatenation of tables from [MMB15, DvDF+16]. Here, $B$ is the block size, $\lambda$ is the security parameter and $n$ is the total number of blocks.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Block Size</th>
<th>Com.</th>
<th>Server comp.</th>
<th>Asmptn</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[AKST14]</td>
<td>$\Omega(\text{polylog }n)$</td>
<td>$O(1)$</td>
<td>$O(\text{polylog }n)$</td>
<td>FHE</td>
<td>SH</td>
</tr>
<tr>
<td>[MMB15]</td>
<td>$\Omega(\log^4 n)$</td>
<td>$O(1)$</td>
<td>$O(B\lambda \log n)$</td>
<td>LWE/DCR</td>
<td>M</td>
</tr>
<tr>
<td>DvDF+16</td>
<td>$\Omega(\log^6 n)$</td>
<td>$O(1)$</td>
<td>$O(B\lambda \log n)$</td>
<td>HE</td>
<td>M</td>
</tr>
</tbody>
</table>

Subsequently, Onion-ORAM [MMB15] was a construction that achieved again constant communication complexity but used as an assumption the Learning with Errors (LWE) [Reg09] or the decisional composite residuosity assumption (DCR) [Pai99] and the security was in the malicious setting.

The idea behind this construction is to give to the server a vector of $\log n$ encryptions of 0 or 1, where 1 will appear only once and will correspond to the node/bucket where the needed block resides. In particular, each of the $n$ blocks $M[i]$ is encrypted $\ell$ times $eb_i = \text{Enc}^\ell(M[i])$ and is stored in its corresponding bucket. When the client wants to retrieve the block $M[i^*]$, it sends the vector $\vec{c}$ where $c_i = \text{Enc}^{\ell+1}(b_i)$, $b_{i^*} = 1$ and $b_{i^*i} = 0$. Then, the server computes the value $ct = \bigoplus_i c_i \cdot eb_i$ which by the homomorphic property of the underlying encryption scheme results to a ciphertext in $\text{Enc}^{\ell+1}(M[i^*])$.

Recently, Devadas et al. [DvDF+16] managed to achieve better parameters in terms of minimum block size and at the same time assuming just additively homomorphic encryption instead of LWE or DCR. The key difference lies in the so called “oblivious bucket merging technique”. During the eviction process when one bucket has to be merged with the bucket below,
the client sends to the server the homomorphic encryption of a permutation \(\pi\) that allows the server to rearrange the blocks within the bucket and subsequently merge them with the child bucket. The permutation is chosen in such a way that the indexes of the real blocks in the final arrangement of the parent bucket contain no real blocks in the child bucket and thus the merging can succeed without any overwriting.

### 3.3 Parallel ORAM

The classical constructions of ORAM fail to naturally extend to the case of parallel computation. In parallel computation a lot of CPUs perform a parallel computation by accessing the same memory. In this case, the following problem appears. If two CPUs try to access the same block then the memory will notice this. In particular it will be able to distinguish such a computation from a computation where each processor accesses different data. Boyle et al. \cite{BCP16} is the first scheme to the best of our knowledge that tries to address the problem of OPRAM achieving better parameters than just serializing the computation and running it sequentially. In particular, they present a generic compiler that takes as input a PRAM and outputs an OPRAM which achieves only a polylogarithmic slowdown in terms of parallel and total complexity, where parallel complexity is defined as the time complexity of the slowest processor. The construction is a tree-based construction (see \cite{SCSL11} and \ref{sec:parallel-oram} for more details). The main issue that needs to be dealt with is the scenario where two or more processors are trying to access the same block in the RAM. Notice that relocating the block every time it is read is not possible now, since the other processor will not know where to look for it. In order to avoid this problem the idea is to coordinate the processors so that they pick one representative which
will actually access the desired block and broadcast its value to the rest of the processors. The remaining processors that are interested in that block perform dummy accesses. Another issue comes up with writing; i.e. having $m$ processors trying to write on the root node will unavoidably cause overflow. [BCP16] deals with this problem by having the writing process not take place on the top level of the tree but rather on the $\log m$’th level, where $m$ is the number of processors. Therefore, the memory can be thought of as a forest of $\log m$ trees, each having height $\log n - \log m$. The $i$th tree is then assigned to the $i$th processor and this processor is responsible for reads and writes to this tree. When a block needs to be written in the memory, a random path is associated with it, and the block is obliviously communicated to the processor that is responsible for this path. Finally, the eviction procedure takes place in parallel: for each node to be flushed, a representative processor is assigned to it which is responsible of doing the appropriate operations. In terms of efficiency, their scheme has a communication overhead of $\omega(\log^3 n)$. Moreover, this construction deals with a variable number of processors activated in each step.

In a followup work, Chen et al. [CLT16] manage to drop the communication overhead to $O(\log^2 n)$ and thus achieving efficiency which is almost as good as the state of the art ORAM schemes. As a drawback, the scheme cannot tolerate a variable number of processors without a serious computational slowdown (see table 3).

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Slowdown</th>
<th>Variable CPU number</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>[BCP16]</td>
<td>$\text{polylog}(n)$</td>
<td>yes</td>
<td>$O(\log^3 n)$</td>
</tr>
<tr>
<td>[CLT16]</td>
<td>$\text{polylog}(n)$</td>
<td>no</td>
<td>$O(\log^2 n)$</td>
</tr>
</tbody>
</table>

Table 3: A comparison between the two available (to the best of our knowledge) schemes for oblivious PRAM that achieve only a polylogarithmic slowdown.
4 Goldreich-Ostrovsky lower bound

Goldreich and Ostrovsky have shown that any classical oblivious ORAM construction (i.e. ORAM without server computation) cannot achieve both $O(1)$ client storage and $o(\log n)$ communication overhead.

**Theorem 1.** Any ORAM with constant client storage has communication overhead at least $\Omega(\log n)$; i.e. communication complexity at least $\Omega(n \log n)$.

**Proof.** We can model the problem as follows. Suppose that the CPU has a constant size memory of $b$ blocks. Suppose, moreover, that the memory has $n$ entries and each of them hosts one of the $n$ blocks. The CPU has a request list $r_1, \ldots, r_t$, with $r_i \in [n]$ and runs in $t$ rounds. At the end of each round $i$, the CPU wants to possess the value of the block $r_i$. During each round the CPU can perform three operations: read a block, write a block or just touch a memory entry without doing anything with it. Moreover, we let $q$ be the total number of interactions with the memory, where $q \geq t$ since in order to obliviously access a block it may need to read, write or touch more. Finally, let $A = (h_1, v_1), \ldots, (h_q, v_q)$ be the access pattern, where $h_i$ is a hidden action specifying whether the CPU reads, writes one of its $b$ blocks or just touches an entry (so in total $h_i$ can take $b + 2$ values) and $v_i$ is a visible action that specifies which of the $n$ entries we apply the hidden action to.

Notice that the hidden action corresponds to the encryption of the blocks and thus the memory does not know what this action is. Therefore, the sequence $A$ can take at most $(b + 2)^q \cdot n^q$ values. On the other hand, in order to achieve obliviousness, the access pattern $A$ must be able to correspond to any possible request list, and in particular to any of the $n^t$. Thus it follows that $(b + 2)^q \cdot n^q \geq n^t$. By setting $n = t$ we get $(b + 2)^q \cdot n^q \geq n^n$ and thus $q \geq n \log n$, which means that we need at least an access pattern $A$ of size
4.1 Tightness of the Goldreich-Ostrovsky lower bound

Wang et al. [WCS15] is the tightest construction that we know that gets very close to the lower bound by achieving overhead $O(\log n)$ assuming blocks of size $\log^2 n$. Although the construction details are quite technical, the proof idea is clear. What they achieve is to reduce the client’s circuit complexity (and thus the communication complexity) by improving the way eviction takes place. In particular, they scan the memory 3 times by keeping only metadata about the blocks and not actually downloading these blocks.

5 Main constructions

All ORAM constructions fall into two categories, the hierarchical ORAM and the tree-based ORAM. In the following we describe in detail these two patterns.

5.1 Hierarchical ORAM

Hierarchical ORAMs, first studied in [GO96] have the following structure. The memory consists of $L = \log n$ buffers $B_k, \ldots, B_L$ each having size $|B_j| = 2^j$. The parameter $k$ is usually a constant. Each of the $2^j$ entries of this buffer does not accommodate only one block but instead it is a bucket that can accommodate at most $b$ blocks (see figure [1]). Each buffer $B_j$ is associated with a random secret hash function $h_j$ and behaves as a standard hash table. In other words, if $B_j$ accommodates the block $M[i]$, then this block will be in the bucket $h_j(i)$ of this buffer. Reading an element $i$ then proceeds as follows: First, the whole buffer $B_k$ is read. Then, for each of the buffers $(B_j)_{j\geq k}$, all $b$ elements of the bucket $h_j(i)$ are read.
and either discarded (if they do not contain $M[i]$) or kept (if they contain it). Of course, even if the block is found early, we continue the search in all buffers so that we avoid leaking any information. After reading it, it is important that this will not remain in the same location, since this would create a correlation and a program that reads the same block twice would be distinguishable from one that does not. The solution to this is to place the block in the top buffer $B_k$. Writing behaves in a similar fashion. When we want to over-write the block $M[i]$, we first perform a dummy read from all buffers so that the adversary cannot distinguish between a read and a write operation, and then we place the block with the new value in the top buffer.

Since all operations involve writing in buffer $B_k$, we need to deal with overflows since every some fixed number of accesses this buffer will be full. In other words, we need to have a procedure $\text{Reshuffle}$ that actually evicts the blocks towards lower buckets. In most constructions $\text{[GMOT12, KLO12]}$ this is achieved as follows. Every $2^i$ accesses, the buffer $B_i$ has to be flushed and all of its elements have to be moved to the lower buffer $B_{i+1}$. For buffer $B_{i+1}$ now, we first pick a new hash function $h_{i+1}$ and store the blocks using this function. This reshuffling procedure, guarantees the invariant that each element is never searched in the same buffer with the same hash function, since every time we read it it goes to the top buffer and if it is evicted to a lower buffer, this buffer is assigned a new hash function.

5.1.1 Cuckoo hashing

Cuckoo hashing, first introduced by Pagh and Rodler $\text{[PR01]}$ is an important technique that has been used a lot in hierarchical based ORAMs $\text{[GMOT12, KLO12]}$. The idea is the following. A buffer $B$ instead of being a standard
Figure 1: Hierarchical ORAM example. Here we have $k = 2$ and so the first buffer has $2^2$ buckets and $L = 5$ so the last buffer has $2^5$ buckets. The size of each bucket is $b = 4$.

hash table, it is associated with two hash functions $h_1, h_2$. A block $M[i]$ may be living either in its first location $h_1(i)$ or its second location $h_2(i)$. Whenever we try to store a block $M[i]$ in $B$, we store it in its first location if it is not already occupied. But if this entry is occupied by a block $M[j]$ then this blocked is kicked out and goes to its other location, possibly kicking-out another element which will have to be moved to its other location etc. For an illustration see figure 2.

In order to deal with overflows in cuckoo-hash tables, a stash of constant size $s$ can be introduced. [KMW09] has shown that in this case the failure probability can drop down to $O(n^{-s})$. Reading an element $i$ in this case proceeds similarly to the standard hash-tables; First we read the whole top buffer and then for each of the following ones we read the entries $h_1(i), h_2(i)$. As before, even if we find the desired block we continue reading from all buffers. Finally we put back the element in the top buffer. Writing, again, is always preceded by a dummy read operation. Last, reshuffling is performed every $2^i$ steps and then all elements in buffer $B_i$ are evicted to the buffer $B_{i+1}$. On this moment $B_{i+1}$ is also assigned two new hash functions.
Figure 2: Cuckoo hashing example. Suppose that we have the values: $h_1(i) = 10, h_2(i) = 5, h_1(j) = 13, h_2(j) = 5, h_1(k) = 16, h_2(k) = 13$. Suppose also that the initial state of the memory is the one appearing in figure 2a and that we want to insert the block $M[l]$ such that $h_1(l) = 10$. The first figure shows how the existing blocks will be kicked out in order to allow the new block reside in address 10; the arrows show where the next address of the block will be. Figure 2b shows the state of the memory at the end of the insertion.
5.1.2 Hybrid model and modifications

Kushilevitz et al. [KLO12] use both types of buffers in order to improve efficiency. In particular, for constants \( k = \log \log n + 1 \) and \( K = 7 \log \log n \), they choose the buffers \( k, \ldots, K \) to be regular hash tables with bucket size \( b = \log n \) and the buffers \( K + 1, \ldots, \log n \) to be cuckoo hash tables. Another modification in their scheme is that each level \( j \in [K + 1, \log n] \) does not have only one buffer but \( t = \log n \) such buffers, and their size is not just \( 2^j \), but \( 2^j \cdot 2^{j-K} \), and thus their size does not just double but actually increases by a factor of \( 2t \). The key observation about these modifications is that they manage to achieve eviction less often and thus reduce the communication. As of today, this scheme is the most efficient hierarchy-based scheme if terms of communication overhead as shown in table 1.

5.2 Tree-based ORAM

Here we present some important details about tree-based ORAM constructions as were initiated by Shi et al. [SCSL11]. This construction is quite generic as most results follow it with small modifications. An illustration of it appears in figure 3.

The client wants to store to the server an array of blocks \( M[1], \ldots, M[n] \) each of which has size \( B \). Most of the times \( B = \log^c n \) for some constant \( c \). The memory is structured as a binary tree of height \( \log n \) so it has exactly \( n \) leaves. Each node of the tree is a bucket that usually can host a constant \( Z \) number of blocks. Each block \( i \) is associated with a randomly chosen by the client path \( p(i) \) in this tree. The main invariant of this construction is that a block can only live in a bucket that belongs to its path. When the client wants to read the block \( i^* \), it looks in its local database to find the associated path in the memory. Then, it queries one-by-one all the buckets in this path.
by discarding any block that is not $M[i^*]$ and just keeping the block $M[i^*]$. After reading this block, the client needs to store it back to the memory. Of course, putting it back to the same position would leak information in the case where it wants to read the same block again. The way to avoid this attack and to achieve security is by having the client re-encrypting the block and putting it back in the top node. Writing a block proceeds almost the same way. However, in order to avoid distinction between reads and writes, every write operation on the top node must be preceded by a dummy read operation.

5.2.1 Eviction

An issue that needs to be addressed is overflow. For example, the top bucket cannot host more than $Z$ blocks, so we need to have a procedure to evict blocks lower in their paths. Several different suggestions for eviction have been proposed and in some cases the eviction procedure is the one causing better of worse efficiency. An idea that is used a lot in recent schemes is the following [CP13]. Every $Z$ accesses to the memory, the client picks a random path in the tree and evicts all the blocks in that tree towards the lowest possible level they can get in that tree. This way, the root bucket is never overflown. [DvDF+16] has proposed another clever way to pick the to-be-evicted paths. Instead of picking them at random, pick them in reverse lexicographic order; e.g. first evict path 00, then 10, then 01 and then 11. This guarantees that the eviction is spread more uniformly in the tree. Moreover, it guarantees that a bucket at level $k$ will be evicted exactly every $2^k$ accesses, thus the client always knows when a bucket is empty. Another idea, used by Shi et al. [SCSL11], is to pick at random two nodes from each level and evict their contents to their appropriate children.
Figure 3: Example of a tree-based ORAM structure with bucket size $Z = 3$ and $n = 8$ blocks. Block $M[4]$ is associated with the path 010.

5.2.2 Recursion

Notice that in this construction, the client needs to store a table that indicates the path associated with each block. A path has size $\log n$ and thus the client has to store locally $n \cdot \log n$ data, instead of $n \cdot B$ data. Therefore, for $B = c \log n$ and $c > 1$, this construction saves a constant factor of $c$. In order to avoid storing this data locally, another tree-based ORAM is invoked and this continues recursively. Finally, after $\log n$ such memories, the client is only holding a constant amount of data locally and invokes all $\log n$ memories in order to retrieve the desired block. Since the biggest memory dominates the complexity parameters, it is enough to evaluate the parameters in this final memory.

5.3 Improving Round Complexity

The tree-based structure that all current constructions are based on, have a special property. Storing $n$ blocks in the memory requires also a local storage of indices of size $n/c$ for some constant $c$ (see section 5.2.2 for details). In order to avoid storing this data locally and achieve only constant local storage, ORAMs apply the same idea recursively, resulting in a sequence of
memories whose total size is dominated by the last one. As a result, when
the client wants to access a particular block, it has to iteratively access each
individual memory which gives a round complexity of $O(\log n)$. Garg et
al. [GMP15], take advantage of garbled circuits in order to avoid multiple
interactions and in particular create **TWORAM**, an ORAM with only two
rounds of communication per query.

In more detail, [GMP15] works on an abstract version of tree-based
ORAMs. As discussed in section 5.2.2 tree-based ORAMs actually con-
sist of $L = \log n$ trees $T_1, \ldots, T_L$ such that each tree $T_i$ has height $\log i$. The
contents of only $T_1$ are actually stored in the client; thus achieving constant
local storage. The association between a block and a path in $T_L$ is actually
stored in a path in $T_{L-1}$. In order to find this path the client needs to access
a path in $T_{L-2}$ and so on. Therefore, the client starts by reading its local
tree $T_1$ which returns a path to $T_2$. Then it reads that path in $T_2$ which
returns an encryption of a path in $T_3$. The client decrypts that value in
order to find which path it has to explore in $T_3$. [GMP15] avoids this whole
interaction by modifying what the nodes of the trees actually store. In par-
ticular instead of storing buckets that contain encryptions of blocks they
actually contain a garbled circuit. This garbled circuit has the encrypted
blocks hardwired in it. Evaluating this garbled circuit with a garbled input
$x$ returns as output a garbled input for the next garbled circuit which will
be either the left or the right child of this node. At the end of the evaluation
path, the path to the next tree is returned.

Eviction takes place by having the client actually creating new garbled
circuits for each of the nodes that will participate in the eviction process.
This does not increase the round overhead since the client can attach this
as part of its query message.
Table 4: A non-exhaustive list of constructions with only two rounds of interaction per access copied partly from [GMP15]. The last column corresponds to the worst case overhead. Here again, $n$ is the total number of blocks.

### 5.4 Parallel ORAM

Here we describe in more detail the Oblivious Parallel RAM scheme by Chung et al. [CLP14]. As we described in the section 3.3, the construction is basically a compiler of an insecure $\Pi$ program to a secure (oblivious) $\tilde{\Pi}$ program, such that the computational overhead is $O(\log(m) \log^3(n))$, where $n$ is the memory size and $m$ is the number of processors. The compiler basically creates a sequence of $\log n$ tree-based ORAMs $T_1, \ldots, T_{\log n}$ such that $T_t$ has depth $2^t$, for $t \in [\log n]$ (see also 5.2.2). Below we describe the algorithm that each CPU $i \in [m]$ runs in order to access the block $r$ with value $v$; where $v = \bot$ if it is a read access and $v \neq \bot$ if it is a write access with value $v$.

**Access**($r, v, t$) (run by CPU $i$)

1. If $t > \log n$ return.
2. Broadcast $(r, v)$.
3. Receive $(r_1, v_1), \ldots, (r_m, v_m)$.
4. Let $\text{rep}(r) = \min_{i \in [m]} \{i : r_i = r\}$ (the lexicographically smallest CPU who is interested in block $r$) and $v' = v_{\text{rep}(r)}$.
5. Recursively call Access in the next tree: If $i = \text{rep}(r)$ then pick a random leaf $\ell'$ and run $\ell' \leftarrow \text{Access}(r, \ell', t + 1)$. $\ell'$ will be the new leaf assigned to block $r$ in tree $T_t$. If $i \neq \text{rep}(r)$ then run $\text{Access}(1, \bot, t + 1)$.
which corresponds to a dummy read.

6. Read path: If $i = \text{rep}(r)$ then read all buckets in the path $\ell$ and keep the one needed: $(r, v_{\text{old}})$.

7. Remove old data:

   (a) If $i = \text{rep}(r)$ then broadcast $(r, \ell)$, else broadcast $(\bot, \ell)$
   
   (b) Remove block $r$ from the bucket.

8. Insert new data:

   (a) If $i = \text{rep}(r)$ then broadcast $(r, \ell', v')$.
   
   (b) Receive $(r_1, \ell'_1, v'_1), \ldots, (r_m, \ell'_m, v'_m)$ and keep only those whose path $\ell'$ belongs to the subtree manipulated by $i$. Insert these blocks in the root of this subtree.


   (a) Pick a random leaf $\ell_f$ and broadcast it
   
   (b) Receive all leaves associated with $i$ and flush them. If there is a conflict with another CPU on some node, the CPU with smallest index will flush that node.

10. If $\text{rep}(r) = i$ broadcast the old value $v_{\text{old}}$.

    The key points of the previous algorithm are the following: Only one CPU will read a specific block even if more CPUs need it. This will be the CPU with the smallest ID that is interested in that block. Then it will broadcast the value to the others. Moreover, since each CPU is responsible for a subtree of the whole tree, it will be the one that is writing back the new value to a node that belongs to that subtree. Note, that if PRAM, if two CPUs want to write to the same block, the value of the CPU with the smallest ID will be written to that block. Flushing/eviction takes place by having each CPU picking a random leaf and broadcasting it. A CPU collects all leaves that correspond to its subtree and flushes them.
Theorem 2. The above construction has $O(\log(m) \log^3(n))$ worst case computational overhead and $\omega(1)$ memory overhead.

6 Conclusion and further directions

The area of ORAMs has become highly active recently with a lot of important results in terms of efficiency and practicality. Moreover, secure cloud computing has made the need for efficient ORAMs more prevalent than any other time. In this survey we presented several results and some of them have managed to get very close to the famous lower bound of Goldreich and Ostrovsky. An interesting open problem in the classical setting of ORAMs would be to construct a scheme with communication overhead of $O(\log n)$ even for blocks of size $\Omega(\log n)$; note that Wang et al. [WCS15] achieve such efficiency but only for blocks of size $\Omega(\log^2 n)$.

Boyle et al. [BCP16] initiated the area of parallel ORAM, where a lot of clients perform a parallel computation by having access to a shared memory. Note that in this case, all the processors are considered honest and security holds only against a malicious RAM. An interesting direction would be to study a scenario where some of the processors behave maliciously and try either to modify the computation or learn data. Although a formal and elegant definition lacks for this, we are optimistic that this is a promising area with a lot of applications in practice.

Other possible lines of research would be the following. Consider the scenario where the client stores its data to a server that does not have enough storage and thus uses another server to store part of this data. What kind of efficiency and security guarantees can we achieve? Moreover, comparing this area with the computational complexity area of Multi-Prover Interactive

---

1Questions in this paragraph came from discussions with Konstantinos Pouliasis.
Proofs (MIP), consider a scenario where the client can use more than one servers that do not communicate with each other. Can we achieve better efficiency in this case?
References


