Perfectly Secure Oblivious RAM Without Random Oracles

Ivan Damgård, Sigurd Meldgaard, Jesper Buus Nielsen

Department of Computer Science, Aarhus University

Abstract. We present an algorithm for implementing a secure oblivious RAM where the access pattern is perfectly hidden in the information theoretic sense, without assuming that the CPU has access to a random oracle. In addition we prove a lower bound on the amount of randomness needed for implementing an information theoretically secure oblivious RAM.

1 Introduction

In many cases it is attractive to store data at an untrusted place, and only retrieve the parts of it you need. Encryption can help to ensure that the party storing the data has no idea of what he is storing, but it may still be possible to get information about the stored data by analyzing the access pattern.

A trivial solution is to access all the data every time one piece of data is needed. However, many algorithms are designed for being efficient in the RAM-model, where access to any word of the memory takes constant time, and so accessing all data for every data access gives an overhead that is linear in the size of the used memory.

This poses the question: is there any way to perfectly hide which data is accessed, while paying a lower overhead cost than for the trivial solution?

Goldreich and Ostrovsky [6] solved this problem in a model with a secure CPU that is equipped with a random oracle and small (constant size) memory. The CPU runs a program while using a (large) RAM that is observed by the adversary. The results from [6] show that any program in the standard RAM model can be transformed using an "'oblivious RAM simulator" into a program for the oblivious RAM model, where the access pattern is information theoretically hidden. The overhead of this transformation is polylogarithmic in the size of the memory.

Whereas it is not reasonable to assume a random oracle in a real implementation, Goldreich and Ostrovski point out that one can replace it by a pseudo-random function (PRF) that only depends on a short key stored by the CPU. This way, one obtains a solution that is only computationally secure. Moreover, in applications related to secure multiparty computation (see below), one would need to securely compute the PRF, which introduces a very significant overhead.

It is a natural question whether one can completely avoid the random oracle/PRF. One obvious approach is to look for a solution that uses a very small

number of random bits. But this is not possible: in this paper we show a lower bound on the number of secret random bits that an oblivious RAM simulator must use to hide the access pattern information theoretically: the number of random bits used must grow linearly with the number of memory accesses and logarithmically with the memory size. The natural alternative is to generate the random bits on the fly as you need them, and store those you need to remember in the external RAM. This assumes, of course, that the adversary observes only the access pattern and not the data written to the RAM. However, as we discuss below, there are several natural scenarios where this can be assumed, including applications for secure multiparty computation. The advantage of this approach is that it only assumes a source that delivers random bits on demand, which is clearly a more reasonable assumption than a random oracle and much easier to implement in a secure multiparty computation setting.

Using this approach, we construct an oblivious RAM simulator where we can make an access with an amortized $\log^3(N)$ overhead, where N is the size of the memory provided. The result remains the same, even if the adversary is told which operations the program is executing, and even if the simulator has no internal memory.

In recent concurrent and independent work [2] Ajtai also deals with oblivious RAM and unconditional security. His result solves essentially the same problem as we do, but using a completely different technique that does not lead to an error-free solution. In Ajtai's algorithm, a certain error event must not occur and Ajtai shows that this event happens with probability only $n^{-\log n}$. If the error event happens, then the simulation reveals information it shouldn't, so one cannot get an error-free solution from Ajtai's by, for instance, rebuilding the entire data structure if things go wrong. In comparison, our algorithm fails with probability zero and is simpler to describe and to prove, being a more "direct fit" to the model.

In work following ours [4], Beame and Machmouchi prove that an oblivious RAM simulation always requires a superlogarithmic overhead in both time and space, but in contrast to our work, their result says nothing about the amount of randomness needed.

2 Applications

Software protection: This was the main original application of Goldreich and Ostrovsky. A tamper-proof CPU with an internal secret key and randomness could run an encrypted program stored in an untrusted memory. Now using an oblivious RAM, the observer would only learn the running time and the required memory of the program, and nothing else. Note that, while the adversary would usually be able to see the data written to RAM in such a scenario, this does not have to be the case: if the adversary is doing a side channel attack where he is timing the memory accesses to see if the program hits or misses the cache, he is exactly in a situation where only information on the access pattern leaks, and our solution would give unconditional security.

Secure multiparty computation: If secure multiparty computation is implemented by secret sharing, each player will have a share of the inputs, and computations can be done on the shares. We can use the oblivious RAM model to structure the computation by thinking of the players as jointly implementing the secure CPU, while each cell in the RAM is represented as a secret shared value. This is again a case where an adversary can observe the access pattern (since the protocol must reveal which shared values we access) but not the data[1]. Using an oblivious RAM, we can hide the access pattern and this allows us, for instance, to do array indexing with secret indices much more efficiently than if we had used the standard approach of writing the desired computation as an arithmetic circuit.

Note that players can generate random shared values very efficiently, so that our solution fits this application much better than an approach where a PRF is used and must be securely computed by the players.

Cloud computing: It is becoming more and more common to outsource data storage to untrusted third parties. And even if the user keeps all data encrypted, analysis of the access patterns can still reveal information about the stored data. Oblivious RAM eliminates this problem, leaving the untrusted party only with knowledge about the size of the stored data, and the access frequency.

3 The model

An oblivious RAM simulator is a functionality that implements the interface of a RAM, using auxiliary access to another RAM (the physical RAM). We say that such a simulation securely implements an oblivious RAM, if for any two access patterns to the simulated RAM, the respective access patterns that the simulation makes to the physical RAM are indistinguishable.

To simplify notation we assume that the interface of a RAM has only one operation, which writes a new value to a memory position and returns the previous value.

We model that the identity of the instructions performed by the simulator leak, but not the operands. We assume that the indices of the memory positions updated by the simulation leak, but not the values being retrieved and stored.

A RAM is modeled as an interactive machine behaving as follows:

1. Set $\mathcal{N}[i] = 0$ for $i \in \mathbb{Z}_N$.
2. On each subsequent input $(\texttt{update}, i, v)$ on the input tape, where $i \in \mathbb{Z}_N$ and $v \in \mathbb{Z}_q$, let $w = \mathcal{N}[i]$, set $\mathcal{N}[i] = v$ and output w on the output tape.

We consider lists of form $U = ((\texttt{update}, i_1, v_1), \ldots, (\texttt{update}, i_\ell, v_\ell))$ with $i_j \in \mathbb{Z}_N$ and $v_j \in \mathbb{Z}_q$. Let $\mathrm{IO}_{\mathrm{RAM}}(U) = ((\texttt{update}, i_1, v_1), w_1, \ldots, (\texttt{update}, i_\ell, v_\ell), w_\ell)$ denote the sequence of inputs and outputs on the input tape and the output tape when the interactive machine is run on the update sequence U, where $|U| = \ell$.

[1] This type of application was also already proposed by Goldreich and Ostrovsky.

Formally, an ORAM simulator is a tuple $\mathcal{S} = (\mathcal{C}, N, M, q)$, where $\mathcal{C} = (\mathcal{C}[0], \ldots, \mathcal{C}[|\mathcal{C}|-1])$ is the finite program code where each $\mathcal{C}[j]$ is one of $(\texttt{random}, i)$, $(\texttt{const}, i, v)$, $(\texttt{+}, t, l, r)$, $(\texttt{-}, t, l, r)$, $(\texttt{*}, t, l, r)$, $(\texttt{=}, t, l, r)$, $(\texttt{<}, t, l, r)$, $(\texttt{goto}, i)$ with $i, t, l, r \in \mathbb{Z}_M$ and $v \in \mathbb{Z}_q$, and $N \in \mathbb{N}$ is the size of the simulated RAM, $M \in \mathbb{N}$ is the size of the physical RAM, and $q \in \mathbb{N}$ indicates the word size of the RAMs: the simulated and the physical RAM store elements of $\mathbb{Z}_q$. We require that $q > \max(N, M)$ so a word can store a pointer to any address. We denote the simulated memory by $\mathcal{N} \in \mathbb{Z}_q^N$, indexed by $\{0, 1, \ldots, N-1\} \subseteq \mathbb{Z}_q$. We denote the physical memory by $\mathcal{M} \in \mathbb{Z}_q^M$, indexed by $\{0, 1, \ldots, M-1\} \subseteq \mathbb{Z}_q$.

The simulation can be interpreted as an interactive machine. It has a register $\mathbf{c}$ and a special *leakage tape*, which we use for modeling purposes. The machine behaves as follows:

1. Set $\mathcal{M}[i] = 0$ for $i \in \mathbb{Z}_M$.
2. Set $\mathbf{c} = 0$.
3. On each subsequent input $(\texttt{update}, i, v)$ on the input tape, where $i \in \mathbb{Z}_M$ and $v \in \mathbb{Z}_q$, proceed as follows:
   (a) Set $\mathcal{M}[0] = i$ and $\mathcal{M}[1] = v$.
   (b) If $\mathbf{c} > |\mathcal{C}|$, then output $\mathcal{M}[2]$ on the output tape, append $(\texttt{return})$ to the leakage tape and halt. Otherwise, execute the instruction $C = \mathcal{C}[\mathbf{c}]$ as described below, let $\mathbf{c} = \mathbf{c} + 1$ and go to Step 3b. Each instruction C is executed as follows:
      - If $C = (\texttt{random}, i)$, then sample a uniformly random $r \in \mathbb{Z}_q$, set $\mathcal{M}[i] = r$ and append $(\texttt{random}, i)$ to the leakage tape.
      - If $C = (\texttt{const}, i, v)$, set $\mathcal{M}[i] = v$ and append $(\texttt{const}, i, v)$ to the leakage tape.
      - If $C = (\texttt{+}, t, l, r)$, set $\mathcal{M}[t] = \mathcal{M}[l] + \mathcal{M}[r] \bmod q$ and append $(\texttt{+}, t, l, r)$ to the leakage tape. The commands `-` and `*` are handled similarly.
      - If $C = (\texttt{=}, t, l, r)$, set $\mathcal{M}[t] = 1$ if $\mathcal{M}[l] = \mathcal{M}[r]$ and set $\mathcal{M}[t] = 0$ otherwise, and append $(\texttt{=}, t, l, r)$ to the leakage tape.
      - If $C = (<, t, l, r)$, set $\mathcal{M}[t] = 1$ if $\mathcal{M}[l] < \mathcal{M}[r]$ as residues in $\{0, \ldots, q-1\}$ and set $\mathcal{M}[t] = 0$ otherwise, and append $(\texttt{<}, t, l, r)$ to the leakage tape.
      - If $C = (\texttt{goto}, i)$, let $\mathbf{c} = \mathcal{M}[i]$ and append $(\texttt{goto}, i, \mathbf{c})$ to the leakage tape.

By $\mathrm{IO}_{\mathcal{S}}(U)$ we denote the random variable describing the sequence of inputs and outputs on the input and output tapes when $\mathcal{S}$ is executed as above on the update sequence U, where the randomness is taken over the values r sampled by the `random`-commands. By $\textsc{Leak}_{\mathcal{S}}(U)$ we denote the random variable describing the outputs on the leakage tape.

Definition 1. *$\mathcal{S}$ is an ϵ-secure ORAM simulator if for all update sequences U, the statistical difference $\Delta(\mathrm{IO}_{\mathcal{S}}(U), \mathrm{IO}_{\mathrm{RAM}}(U)) \leq \epsilon$ and it holds for all update sequences U_0 and U_1 with $|U_0| = |U_1|$ that $\Delta(\textsc{Leak}_{\mathcal{S}}(U_0), \textsc{Leak}_{\mathcal{S}}(U_1)) \leq \epsilon$. We say that $\mathcal{S}$ is a perfectly secure ORAM simulator if it is a 0-secure ORAM simulator.*

4 Oblivious sorting and shuffling

In the following, we will need to shuffle a list of records obliviously. One way to do this, is to assign a random number to each, and sort them according to this number. If the numbers are large enough we choose distinct numbers for each value with very high probability, and then the permutation we obtain is uniformly chosen among all permutations.

This issue is, in fact, the only source of error in our solution.

If we want to make sure we succeed, we can simply run through the records after sorting to see if all random numbers were different. If not, we choose a new random set of numbers and do another sorting. This will succeed in expected $O(1)$ attempts each taking $O(n)$ time, and so in asymptotically the same (expected) time, we can have a perfect solution.

We can sort obliviously by means of a sorting network. This can be done with $O(n \log n)$ compare-and-switch operations, but a very high constant overhead [1], or more practically with a Batcher's network [3] using $O(n \log^2 n)$ switches.

Each switch can be implemented with two reads and two writes to the memory, and a constant number of primitive operations. Sorting in this way is oblivious because the accesses are fixed by the size of the data, and therefore independent of the data stored.

By storing the choice bits of each switch we can arrange according to the inverse permutation by running the elements through the switches of the sorting network in backwards order while switching in the opposite direction from before.

5 A solution with polylogarithmic overhead

Like the original algorithm of Goldreich and Ostrovsky in [6], the algorithm works by randomly permuting the data entries, and then storing the permutation in a dictionary data structure used for lookups. Previously accessed elements are put into a cache to ensure we do not have to look at the same place twice, as that would reveal patterns in the accesses. The dictionary also contains dummy elements we can look at, to hide whether we found an element in the cache or in the dictionary. We amortize the time used for searching the increasingly large cache by making levels of caches of growing sizes that are reshuffled when they have been used enough.

The construction in [6] used a hash-table as the dictionary, and security was proven in the random oracle model. A random oracle allows to remember and randomly access a large number of random bits "for free", but we want to do without this, so instead we save our randomness in physical memory using a binary tree for the dictionary.

We now present a solution with an amortized overhead per access of $\log^3 N$. The solution proceeds somewhat like the protocol of [6] with polylogarithmic overhead.

As in [6] the idea is to have $\log_2(N) + 1$ levels of simulated RAMs each level being a cache for the one below. The cache at level 0 has size 1, and the one at level $\log_2(N)$ size N, and in general the cache at level ℓ stores $O(2^\ell)$ elements.

As in [5] it is also possible to make a conceptually simpler solution with a higher asymptotic overhead, this might be useful for some input sizes, and also may aid in the understanding of the full algorithm, we describe that in section 6.

5.1 Shuffled trees

A shuffled tree is a complete binary search tree with all nodes (except the root node) permuted randomly in the RAM. So when you follow a child-pointer, it will give you a uniformly random location in the RAM where the child-node is stored. The actual data is stored at the leafs.

Deciding between the left and right child can be done obliviously by looking at both pointers. In this way a full lookup in a shuffled tree is oblivious: an adversary observing the access pattern to the RAM storing the shuffled tree, sees only $\log N$ random accesses to nodes, and thus learns nothing about what element we find.

In the algorithm described below there is a single logical tree. The nodes of this tree are represented by records in physical memory.

We will have several levels of caches, each can store a number of records representing nodes from the tree. Pointers in the tree will be represented as a pair: the number of the level and an offset into the memory block for that level. The division of the physical RAM into levels is public information that does not have to be hidden.

During a lookup we look at some nodes from the logical tree, this results in touching the records representing these nodes. However, we will never again read those records, we now call them *dead* records. Instead new records representing the same nodes of the tree will be created on lower levels. In this way a node of the tree might be represented by several records on different levels, but only one will ever be alive. Specifically every logical node will always be represented by a single live record at some level. In section 5.3 we show how to construct such a tree obliviously.

5.2 Overview of the construction

The main ideas behind the solution are as follows:

Initial setup We start with a random shuffled tree at level $\log N$ with all data in this tree and all other levels (the caches) being empty.
Each internal node of the tree stores: a left and a right child-pointer, each storing both the level, and the index of that level where the child can be found, a bound for dispatching during the lookup (this bound can actually be inferred from the path we are following, and is mostly included for easing the presentation), and a tag telling if it is a dummy node or not. The leaf nodes store a data-item and the original index of the item. The smallest kind of node has to be padded to make them have the same size.

Lookup and caching When a lookup is made, we start at the root node, and follow child-pointers to a leaf, like when looking up in a binary tree. But now a child pointer can point to a node at the same or any higher cache-level.

When a path $(\nabla_1, \ldots, \nabla_m)$ is followed, we first try to put the nodes $\{\nabla_i\}$ in the bottom level, i.e. level 0. If this level is empty, the path is shuffled (such that the nodes ∇_i are stored in random positions at level 0 and such that ∇_i points to the new physical location of ∇_{i+1} at the bottom level) and inserted here. Otherwise it is merged with the tree already on this level, and we try to put the result on the next level and so on until we find an empty level. Below we describe a procedure for merging two trees while shuffling them so all records end up being permuted randomly.

The pointers from ∇_i to the physical addresses of the siblings of the ∇_i which were not on the path $(\nabla_1, \ldots, \nabla_m)$, and hence were not moved, are just copied along with the nodes during the shuffling, so that the nodes ∇_i at the bottom level might now have pointers to untouched nodes in the trees at higher levels.

The root The only node whose true position is not hidden is the root node. Because it is touched as the first node for every look-up, it will always be at the first position of the youngest tree. We take special precautions to make sure it is not moved during the shuffling procedure. This is still oblivious because the node is touched for every lookup independently of the index we search for.

Dummy nodes If we were just to start each search at the root node and then following the updated points to physical addresses it is true that we would never touch the same node twice in the same position, as a touched node is moved down and then shuffled up. The pattern of how the path jumps between the levels would however leak information.[2] We therefore make sure to touch each level once every time we follow one pointer by doing *dummy reads* at all other levels. If we are following a pointer to level ℓ we do dummy read at levels $i = \log_2 N, \ldots, \ell + 1$, then we read the node at level ℓ, and then we do dummy reads at levels $i = \ell - 1, \ldots, 1$. Only the real node ∇ read at level ℓ is moved to the bottom level.

This requires the tree at each level to contain as many dummy nodes as it has real nodes. These will be chained by their child-pointers, so we can chose for each level to either look at the real node or the next dummy. The location of the head of the dummy chain for each level is stored publicly, like the root node.

[2] If, e.g., we look up the same leaf node twice, the path to the node is moved to the bottom level in the first update, so the second update would only touch nodes at the bottom level. If we look up a distinct leaf in the second update the pointer jumping would at some point take us back to the bottom level. This would allow the adversary to distinguish.

Algorithm 5.1: LOOKUP(key)

output: Value stored in the ORAM at index key
global: $live$ — The set of levels containing records
($current\text{-}level$, $current\text{-}offset$) $\leftarrow$ ($\min(live), 0$) — Start at root
for $i \leftarrow 0$ **to** $\log_2(N)$
do
- **for each** $level \in live$
  - **do**
    - **if** $level = current\text{-}level$
      - **then**
        - $\nabla \leftarrow physical[level, current\text{-}offset]$
        - **if** $key < \nabla.bound$
          - **then**
            - ($next\text{-}level$, $next\text{-}offset$) $\leftarrow$ ($\nabla.left.level, \nabla.left.offset$)
            - ($\nabla.left.level, \nabla.left.offset$) $\leftarrow (0, i)$
            - $physical[level].dummy \leftarrow physical[level].dummy$
          - **else** {. . . same thing for right side. . .
      - **else**
        - Dummy lookup:
        - $t \leftarrow physical[level, physical[level].dummy]$
        - $physical[level].dummy \leftarrow t.left.offset$
  - $path \leftarrow path \cdot \nabla$
  - ($current\text{-}level$, $current\text{-}offset$) $\leftarrow$ ($next\text{-}level$, $next\text{-}offset$)

INSERTPATH($path$)
return ($\nabla.data$)

Algorithm 5.2: INSERTPATH($path$)

global: $live$ — The set of levels containing records
$level \leftarrow 0$
while $level \in live$ — Cascade down until we find a not-live level
do
- $path \leftarrow$ MERGELAYERS($path, physical[level]$)
- $live \leftarrow live - \{level\}$ — Level is no longer live
- $level \leftarrow level + 1$

$physical[level] \leftarrow path$ — Insert the merged nodes
$live \leftarrow live \cup \{level\}$

A sequence of accesses is illustrated in Fig. 1. The procedure `MergeLayers` is not described in pseudocode, but it is explained in detail in the following section.

5.3 Merging and shuffling of two unbalanced trees

Each time a lookup is made, we build a chain of records representing nodes on a path from the root of the tree to the leaf-node we are looking up by copying the nodes from the used path, and making sure that the child pointer we follow points to the next node.

This chain will go to the bottom level cache. If the bottom level already is filled, it will be merged with the new one path and they will be put one level higher, etc.

The records we merge from two levels represent some subset of the nodes from the logical tree. This subset is always forming a tree but not in general a full tree, but an unbalanced tree.

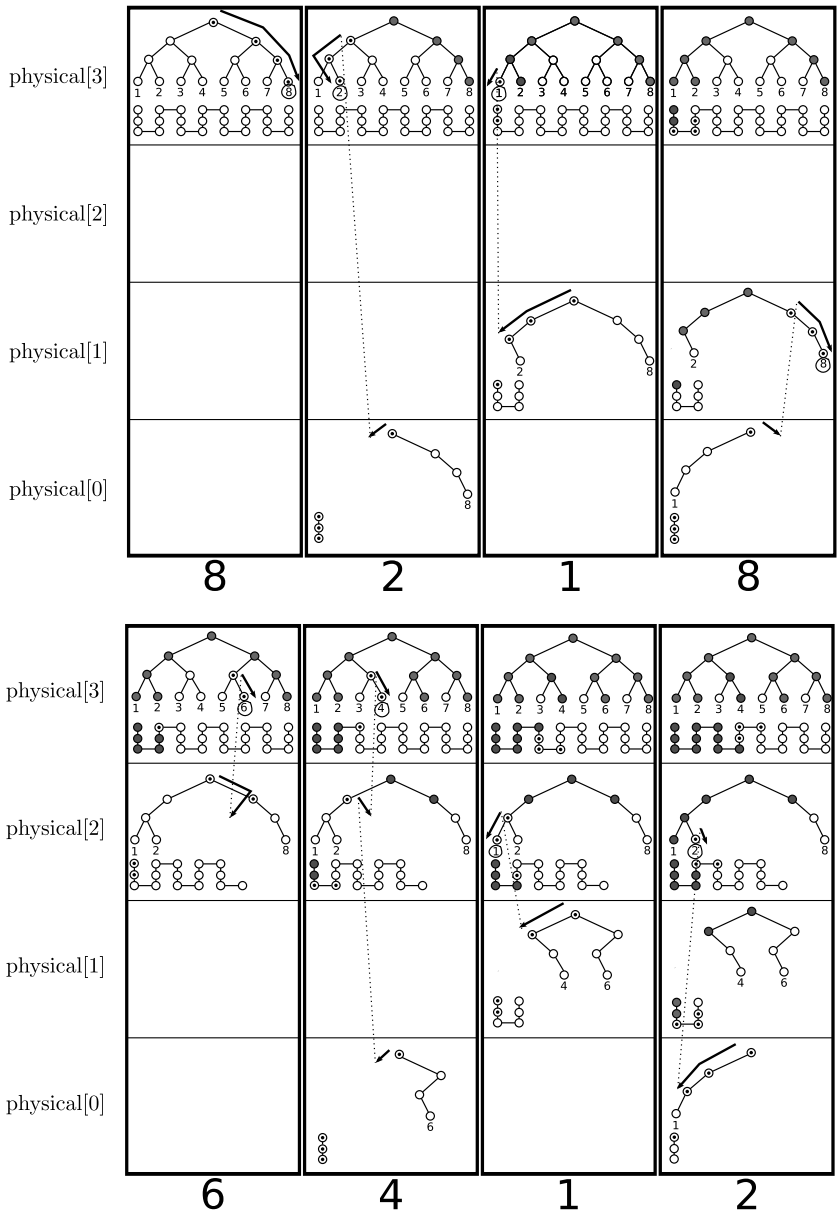


Fig. 1: To be read as a comic strip. Each frame shows the logical (unshuffled) view of the different layers of each step of a sequence of accesses to an oblivious RAM storing 8 elements. Dummy elements at each level are shown at the bottom. The arrow shows how the lookup progresses, and nodes with a dot are those read at this step. Used nodes are shown in gray. Notice how no node is ever touched twice, and how, besides the root node, $\log n$ nodes are read from each live level per lookup.

We now describe how to merge and shuffle two such trees while updating the pointers so they point to the right new place.

We make a crucial observation about the algorithm; that the two merged levels are always two youngest existing levels, this in turn gives us the following:

- There will never be pointers from other levels into the merged records so nothing outside the level has to be updated.
- The child-pointers that are not internal to the two levels will point out of the local level and into an older level, as older levels are not updated, these pointers do not have to be changed.
- The root of the tree will always be in one of the merged levels (the youngest one).

We will copy all the records from both trees that we have not touched yet (the live records) into a new list in the same order as they where before (but with the dead records removed). We run through the second list, and obliviously add to each internal child pointer the number of live records before it in the list, so all internal child-pointers are unique.

Also we connect the two dummy chains, by running through all the records of the first tree, and for each obliviously checking if it is the end of a dummy chain, if so, it is set to point at the child of the head of the dummy chain of the second tree.

In the two lists there will be a number of live dummies already, and they will be reused, but for the resulting list we need more dummies (as many as there are already), so we make a chain of new dummies and make the end of that point to the head of the old dummy chain. These new records are put at the end of the list of nodes, and we shuffle them together with the rest.

We know that every live node is a unique representative of a node in the logical tree. It always (except for the root) has exactly one incoming pointer from the other nodes in the list. And it has one or two children among the copied nodes, the pointers to these needs to be updated, while the (one or zero) child-pointers pointing *out of the tree* to an older tree from where it was copied should stay unchanged.

The root-node and the head of the dummy chain are exceptions to this invariant as they have no incoming pointers, on the other hand their location is not secret, and we take special care of these two nodes.

Note that we cannot allow the dummy nodes to have both child-pointers pointing to the next node because that would give dummy nodes two incoming nodes as well. Instead they must have one forward pointer, an nil-pointer and a tag so we can recognize them as dummy nodes, and when following such one, we always obliviously choose to follow the real pointer.

We now collect this into a subroutine for updating the pointers while we merge and shuffle the two lists with a total of p nodes at the same time. The process is illustrated in Fig. 2

1. For each child pointer in the list in order, we write in a new list a pointer y that, if the pointer is internal, is a copy of the child pointer and ∞ otherwise

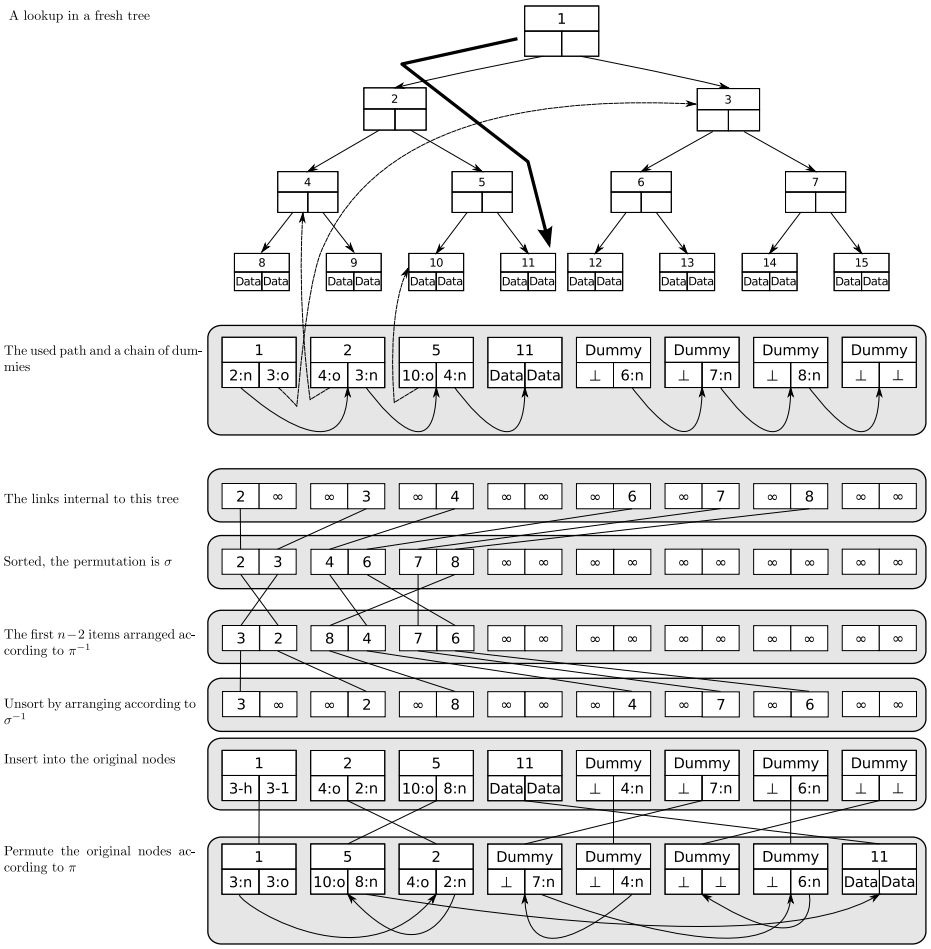


Fig. 2: Illustration of the shuffling process. ":n" here indicates a pointer inside the level that is shuffled, ":o" is a pointer to the old level where the node was copied from.

so it will be sorted last. These represent pointers out of the tree, and should not be updated.
When shuffling a list with p live records, the list of pointers will have $2p$ entries. Approximately half of these will be external pointers, and nil-pointers from dummies. It might be more than half, because the paths represented by the two trees might overlap. This does not affect the shuffling procedure, only the dummy chains will end up longer than what can ever be used before reshuffling, but this knowledge is never revealed to the adversary.

2. Sort the row of y's obliviously, and save the sorting permutation as σ. Because each node (except the two special nodes) has exactly one incoming pointer, the first $p-2$ nodes will be internal, and those are the ones we want to update, so they point to the new locations of those nodes.
3. Now we create the permutation, π, that we want to shuffle the final list in. This permutation should leave location 1 (the root) and the location of the head of the dummy chain the same place, and be uniformly random for all other indices.
4. Now permute a list containing the numbers from 1 to p according to π^{-1}, and remove the first element, and the dummy-head pointer (they stay in place under π). Take this list of $p-2$ elements and concatenate them with $p+2$ ∞'s.
5. Now first undo the sorting by applying σ^{-1} to this list, and use the unsorted list to overwrite the pointers of the original records where they came from (obliviously choosing to change the value only when the new value is $\neq \infty$), now we have updated all the internal pointers, and left the external pointers untouched.
6. Finally shuffle the nodes according to π. We now have a shuffled list with all nodes pointing to the right places.

The main trick behind this procedure is that $\pi(\mathcal{M})[\pi^{-1}(x)] = \mathcal{M}[x]$ (where $\mathcal{M}$ denotes the physical memory, seen as an array). The time the procedure takes is dominated by the time it takes to sort the list of pointers.

5.4 Security

The transcript of a lookup, as seen by the adversary, always consists of the following parts:

- An access to the root node, it will always be the first entry of the youngest tree
- $\log n$ accesses at uniformly random records at each live cache-level of the tree
- A fixed amount of merging and reshuffling, done obliviously

This transcript is independent from the simulated access, and therefore the algorithm is secure.

5.5 Performance

To see how many log factors we use, we look at the two parts of the algorithm (that takes time). That is the operations for doing the *lookup*, and the operations for doing the *reshuffling*.

Following a path of length $\log N$ requires to look up $\log N$ nodes. For each node we have to touch each level for obliviousness, which gives a total of $\log^2 N$ reads.

For reshuffling we saw that the time for shuffling a level is dominated by sorting it. We have $\log N$ cache-levels. Level i is of size $O(2^i \cdot \log N)$, so that takes $O(2^i \cdot \log N \cdot \log(2^i \cdot \log N)) = O(2^i \cdot \log N \cdot i)$ operations to sort (because i is in $O(\log N)$, so it dominates $\log \log N$). But that level is only shuffled every 2^{i-1}th. lookup. On average we therefore use time for all levels:

$$O\left(\sum_{i=1}^{\log N} \log N \cdot i\right) = O(\log^3 N) \ .$$

Therefore the total amortized time spent per lookup is in $O(\log^3 N)$.

This gives rise to the following theorem:

Theorem 1. *The described algorithm implements a perfectly secure ORAM simulation in the standard model with a time and memory overhead in* $O(\log^3 N)$.

6 The square root algorithm

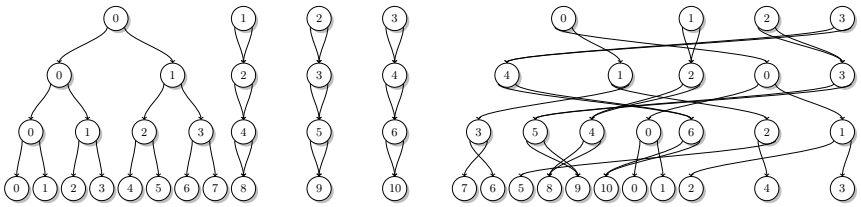
In this section we describe a simpler algorithm implementing an oblivious RAM using memory and amortized time in $O(\sqrt{N} \log^2 N)$. The algorithm assumes access to a functionality that shuffles n elements of the physical RAM in time $O(n \log^2 n)$.

In this solution the tree is stored with each level shuffled individually, so each child pointer from a node points to a random location of the next level. Also we make $\sqrt{N}$ dummy chains, that are also shuffled into the tree. Only the root level is (trivially) not shuffled. The shuffling is depicted in Fig. 3.

Making a lookup A lookup in the simulated RAM is implemented by making a lookup in the binary search tree. In order to touch every node in the tree only once, we do a (possibly dummy) lookup in the physical RAM on each level of the tree, and for each level we also linearly scan through all of the cache to see if we have accessed the same node earlier. If we found the element in the cache, the next access in the tree will still be to a dummy node.

The physical memory is split up into $\log_2(N)$ parts, the i'th part is again split in two; $physical[i]$ storing $2^i + \sqrt{N}$ records (the tree, and the dummy chains), and $cache[i]$ storing $\sqrt{N}$ records (the cache). Each node in the three record stores a .left and .right field for pointing to the next level, and a .bound for directing the search in the tree.

The leaf-records at the lowermost level are different, they contain the .data that are stored, an .index field naming the original index where the data is stored in the simulated RAM, and a .used field that is used for reshuffling the data as described below.



(a) Before (b) After

Fig. 3: Visualization of the memory layout of a tree storing 8 elements, before and after shuffling the tree. The edges indicate child-pointers.

An invariant for the outer loop in the Lookup-algorithm below can be phrased:

1. *next* is the real index we are going to look for at *level*
2. $next\_from\_tree$ is the index of the tree where we will look if we do not find the item in the cache. If this is different from next, it is still pointing at a dummy chain.

By changing the index of the cached value to ∞ when it is found at 1 we implicitly invalidate it; it will be sorted last and therefore thrown away when we reshuffle. This is only necessary to do for the cache of the last level of the tree.

Obliviously shuffling a tree This is a somewhat simpler algorithm for shuffling a tree that the one described in the main part of the paper: it works by shuffling one level at a time, starting at the bottom of the tree, and ending with the root-level. Because we always have all nodes present, we can allow ourselves to consider only one level at a time. After shuffling a level L with permutation π (so $L'[i] = L[\pi(i)]$), we apply π^{-1} on a sequence $[1, \ldots, n]$, and copy these numbers into the child-pointer fields of the level above. This gives the right result, because what before pointed at k will now point at $\pi^{-1}(k)$. But looking up in the shuffled layer yields the record at $L'[\pi^{-1}(k)] = L[\pi(\pi^{-1}(k))] = L[k]$. We take special care to also shuffle the dummy chains, and ensuring that their left and right pointers point to the same next node in the chain.

Algorithm 6.1: UNMINGLE( )

$a \leftarrow$ Filter out any physical[$\log_2 N$] record which has `.used= true`
$b \leftarrow a$ concatenated with cache[$\log_2 N$].
Obliviously sort b according to the original *index* of the records
Remove the last $\sqrt{N}$ records of b
$physical[log_2 N] \leftarrow b$

Algorithm 6.2: SHUFFLE( )

for $level \leftarrow log_2(n)$ **downto** 1
do {
  Choose a permutation π uniformly at random
  Shuffle physical[level] according to π
  $temp = [1, 2, \ldots, 2^{level} + \sqrt{N}]$
  Shuffle temp according to π^{-1}
  for $i \leftarrow 0$ **to** $2^{level-1}$
    do { $physical[level-1, i]$.left $\leftarrow temp[2i]$
             $physical[level-1, i]$.right $\leftarrow temp[2i+1]$
  for $i \leftarrow 2^{level-1}$ **to** $2^{level-1} + \sqrt{N}$
    do { $physical[level-1, i]$.left $\leftarrow temp[2^{level} + i]$
             $physical[level-1, i]$.right $\leftarrow temp[2^{level} + i]$

Algorithm 6.3: DISPATCH($key, record$)

output: The left or right child of record, depending on key
if $key < record.bound$
  then return ($record$.left)
  else return ($record$.right)

Algorithm 6.4: LOOKUP(key)

input: key
output: Value stored at key
if $count \geq \sqrt{n}$
  then { UNMINGLE()
            SHUFFLE()
            $count \leftarrow 0$
  else $count \leftarrow count + 1$
$next \leftarrow count$
$next\_from\_tree \leftarrow count$
for $level \leftarrow 0$ **to** $\log_2(N) - 1$
do {
  $found \leftarrow False$
  for $i \leftarrow 0$ **to** $count - 1$
    do { $k \leftarrow cache[level, i]$
             if k.index = next
               then { $k\_from\_cache \leftarrow k$
                         $found \leftarrow True$
                         k.index $= \infty$
                         $cache[level, i] \leftarrow k$
  if $found$
    then $next \leftarrow next\_from\_tree$
  $k\_from\_tree \leftarrow physical[level, next]$
  $physical[level, next]$.used $= True$
  $next\_from\_tree \leftarrow$ DISPATCH($index, k\_from\_tree$)
  $next \leftarrow$ DISPATCH($index, k\_from\_tree$)
  if $found$
    then $next \leftarrow$ DISPATCH($index, k\_from\_tree$)
  $cache[level, count] \leftarrow (next,$ UPDATE(k))

(1)

Security The transcript of a lookup, as seen by the adversary, always consists of the following parts:

- An access at index *count* of the first level of the tree
- An access at a uniformly random location at each lower level of the tree
- A scan of the full cache of each level
- For each $\sqrt{N}$ accesses, the tree is reshuffled

All these are independent of the access pattern to the original RAM, and thus an eavesdropper will learn nothing whatsoever about the access pattern.

Performance The time for a single lookup (without the shuffling) is dominated by accessing $\log_2 N$ caches, each of size $O(\sqrt{N})$. For each $\sqrt{N}$ lookups, we perform a shuffle taking time $O(N \log^2 N)$. Giving an amortized running time of $O(\sqrt{N} \log^2 N)$ per lookup.

7 Lower bounds on randomness

An equivalent way to state the definition of a secure oblivious RAM simulation is that, for simulation of any program running on an standard RAM, the resulting distribution of accesses to physical memory is the same for every choice of input. In particular, if we choose (part of) the input at random, the distribution of memory accesses must remain the same.

Our goal in this section will be to show a lower bound on the number of random bits that a simulation must use in order to be secure. We will for the moment assume that the simulation is *data-oblivious*, i.e., the simulation treats every word it is asked to read or write as a black-box and does not look at the data it contains. Any such word is called a data-word. All known oblivious RAM simulators are data oblivious. Indeed, letting anything the simulator does depend on the content of a data-word would only seem to introduce extra difficulties, since at the end of the day the access pattern must not depend on this content. Nevertheless, we show in section 8 a weaker lower bound for data non-oblivious simulation.

To show the lower bound, we consider the program that first writes random data to all N locations in RAM. It then executes d read operations from randomly chosen locations. Let U denote this sequence of update instructions.

Let P be the random variable describing the choice of locations to read from. Clearly $H(P) = d \log N$. Let $C = \textsc{Leak}_{\mathcal{S}}(U)$ be the random variable describing the history of the simulation as seen by the adversary. Finally, let K be the random variable describing the random choices made by the simulation during the execution of the program, the r-values of the `random`-commands. We will show a bound on $H(K|C)$, that is, a bound on the number of random bits that are unknown to the adversary.

By construction of the random experiment, K is independent of P and, assuming the simulation is perfectly secure, C and P are independent.

From this it follows by elementary information theory that

$$H(K|C) \geq H(P) - H(P|C,K) = d \log N - H(P|C,K) \ . \tag{1}$$

Now, let us assume that each read operation causes the simulation to access at most n locations in physical RAM. From this we will show an upper bound on $H(P|C,K)$.

Let P_i be the random variable describing the choice of location to read from in the i 'th read. Then we can write $P = (P_d, \ldots, P_1)$, and we have

$$\begin{aligned} H(P|C,K) &= H(P_1|C,K) + H(P_2|P_1,C,K) + \cdots + H(P_d|P_{d-1}..P_1,C,K) & (2) \\ &\leq H(P_1|C,K) + H(P_2|C,K) + \cdots + H(P_d|C,K) \ . & (3) \end{aligned}$$

The plan is now to bound $H(P_i|C = c, K = k)$ for each i and arbitrary, fixed values c, k, from this will follow a bound on $H(P|C,K)$. We will write $H(P_i|C = c, K = k) = H(P_i|c,k)$ for short in the following.

Note first that once we fix $K = k, C = c$, in particular the value of c specifies a choice of at most n locations that are accessed during the i'th read operation. This will constrain the distribution of P_i to be only over values that cause these locations to be accessed. Let w be the number of remaining possible choices for P_i. This is a set of addresses that we call the *relevant* addresses. Since there are w relevant addresses, we have $H(P_i|c,k) \leq \log w$.

Let $a = \log_2 q$ be the number of bits in a memory location, and recall that the program initially writes random data to all addresses. Let the random variable $D_{c,k}$ represent the choice of data originally written to the w relevant addresses. Since the simulation is data oblivious, fixing $C = c, K = k$ does not constrain the data stored in the relevant addresses, and hence $D_{c,k}$ is uniform over the 2^{aw} possible values, or equivalently $H(D_{c,k}) = aw$.

Let $R_{c,k}$ represent all the data the simulator accesses while it executes the i'th read operation given the constraints we have defined. Since at most n words from external memory are accessed, we have $H(R_{c,k}) \leq an$.

But on the other hand, $H(R_{c,k})$ must be at least $H(D_{c,k})$, since otherwise the simulator does not have enough information to return a correct result of the read operation. More precisely, since the simulation always returns correct results, we can reconstruct the exact value of $D_{c,k}$ as a deterministic function of $R_{c,k}$ by letting the value of P_i run through all w possibilities and computing in each case what the simulation would return. Since applying a deterministic function can only decrease entropy, it must be that $H(D_{c,k}) \leq H(R_{c,k})$.

We therefore conclude that $aw \leq an$, and from this and $H(P_i|c,k) \leq \log w$ it follows immediately that $H(P_i|c,k) \leq \log n$. By definition of conditional entropy we have $H(P_i|C,K) \leq \log n$ as well, and hence by (3) that $H(P|K,C) \leq d \log n$. Combining this with (1) we see that $H(K|C) \geq d \log(N/n)$, when d read operations are executed. Thus we have:

Theorem 2. *Suppose we are given a perfectly secure oblivious RAM simulation of a memory of size N. Assume it accesses at most n locations in physical RAM per read operation and is is data-oblivious. Then there exist programs such that*

the simulator must, on average, use at least $\log(N/n)$ *secret random bits per read operation.*

Extensions Suppose the simulation is not perfect, but leaks a small amount of information. This means that P and C are not independent, rather the distributions of C caused by different choices of P are statistically close. Therefore the information overlap $I(C;P)$ is negligible as a function of the security parameter. If we drop the assumption that P, C are independent (3) becomes $H(K|C) \geq d \log N - H(P|C,K) - I(P;C)$. The rest of proof does not depend on C, P being independent, so the lower bound changes by only a negligible amount.

The result also holds even if the adversary does not know which instructions are executed by the simulated program, as the proof does not exploit the fact that in our model, he knows this information.

Another remark is that the result assumes that every read operation causes at most n locations to be accessed. This does not, actually, cover the known solutions because they may at times access a very large number of locations, but do so seldom enough to keep the amortized overhead small. So we would like to show that even if we only assume the amortized overhead to be small, a large number of secret random bits is still needed. To this end, consider the same program as before, and suppose we have a simulation that accesses at most n_i bits during the i'th read operation. Let $\bar{n}$ be the average, $\bar{n} = \frac{1}{d}\sum_i n_i$. Then we have

Theorem 3. *Suppose we are given a perfectly secure oblivious RAM simulation of a memory of size* N*. Assume it accesses at most* n_i *locations in physical RAM during the* i*'th read operation and is data-oblivious. Then there exist programs such that the simulator must, on average, use at least* $\log(N/\bar{n})$ *secret random bits per read operation.*

Proof. Observe first that the argument in the proof for Theorem 2 for equations (3) and (1) still holds here, and that furthermore the argument for $H(P_i|C,K) \leq \log n$ does not depend on the bound n being the same for every read operation. Hence, under the assumption given here, we get $H(P_i|C,K) \leq \log n_i$, and hence by (3), (1) that

$$\begin{aligned} H(K|C) &\geq d \log N - \sum_{i=1}^{d} \log n_i = d \log N - d\frac{1}{d}\sum_{i=1}^{d} \log n_i \\ &\geq d \log N - d \log(\frac{1}{d}\sum_{i=1}^{d} n_i) = d \log N - d \log \bar{n} = d \log(N/\bar{n}) \ . \end{aligned}$$

where the last inequality follows from Jensen's inequality.

8 Data Non-Oblivious Simulation

The argument used in section 7 breaks down if the simulation is not data-oblivious. The problem comes from the fact that then, even if security demands that C is independent of the data D that is written, this may no longer be the case if K is given. And then, if we fix C, this may mean that the data written in the w locations in the proof above are no longer uniformly distributed.

We can nevertheless prove a different lower bound. Since we assume that $q \geq N$ we have that the word size $a = \log_2 q$ is $\theta(\log N)$. Our result would get stronger if a was larger. The goal will be to show that when executing $\theta(N)$ read operations, security implies that $H(K|C)$ must be at least $\theta(N)$, and so one must use a constant number of secret random bits per read operation.

We will consider the same program as in the previous subsection, and we will reuse the notation. We will, however, choose d to be $d = \alpha N$ where α is a constant.

Observe first that since both C and K are independent of D, it follows that $I(K; D|C) = I(C; D|K)$. It is also straightforward to see that $H(K|C) \geq I(K; D|C)$. So if $I(C; D|K) \geq d$ we are already done. So, we may assume $I(C; D|K) < d$. We have that $H(D) = Na$, and also it holds that

$$H(D|C, K) \geq H(D) - I(K; D|C) - I(C; D|K) \geq Na - 2d,$$

where the first inequality holds for any 3 random variables D, K, C. Now by our choice of d, it is much smaller than Na, so $H(D|C, K)$ is almost as large as it can be. So since $H(D|K, C)$ is the average of the values $H(D|K = k, C = c)$ we want to claim that $H(D|K = k, C = c)$ is "large most of the time". Concretely, let p be the probability that $H(D|K = k, C = c) \leq 3Na/4$, taken over the choices of K, C. Then it is straightforward to argue that

$$(1 - p) \cdot Na + p \cdot 3Na/4 \geq Na - 2d = Na - 2\alpha N,$$

from which we conclude that $p \leq 8\alpha/a$, and so is $\theta(1/\log N)$.

We will now do exactly the same argument as for the previous theorem, so we will want to bound $H(P_i|c, k) \leq \log w$, by looking at the number w of possible choices for P_i given that $C = c, K = k$.

We will use the fact we established above, that with probability $1 - p$, $H(D|c, k) \geq 3Na/4$. Values c, k for which this holds will be called a *good* choice of c, k.

So we will assume for the moment that we are in the good case. Now, there are two sub-cases: first, if $w \leq N/2$ we already know enough, as we shall see in a moment. In the second sub-case $w > N/2$, and we now use that $H(D|c, k) \geq 3Na/4$: there are at most $N/2$ words that are *not* among those w we consider, so their entropy can be at most $Na/2$. It follows that the joint entropy of the w words we do consider is at least $Na/4 \geq wa/4$.

We now argue as before, that since the simulator only has access to at most na bits in order to answer the read request, it must be that $na \geq wa/4$, or $w \leq 4n$.

Hence, for a good choice of c, k we have

$$H(P_i|c,k) \leq \max(\log(N/2), \log(4n)).$$

We now make the reasonable assumption that $4n \leq N/2$, and note that even for bad values of c, k $H(P_i|c,k) \leq \log N$. From this we conclude that

$$H(P_i|C,K) \leq p \log N + (1-p)\log(N/2) = \log(N/2) + p \log 2.$$

Plugging this into (1), we finally obtain that

$$H(K|C) \geq d \log N - d \log(N/2) - dp \log 2 = d(1 - p \log 2),$$

which is indeed $\theta(N)$, since p is $\theta(1/\log N)$, and is in fact essentially d for large N. We therefore have the following:

Theorem 4. *Suppose we are given a perfectly secure oblivious RAM simulation of a memory of size N, accessing at most n locations in physical RAM per read operation. Even if the simulation is data non-oblivious, if $n \leq N/8$, such a simulation must use, on average, at least $\theta(1)$ secret random bits per read operation.*

The analysis in fact shows that one must use (essentially) 1 bit per read operation. This bound is almost certainly not tight, but we have not spent much effort in optimizing it, since data oblivious simulation clearly is the way to construct actual solutions.

The result can easily be extended to the case where the simulation does not access the same number of locations in every operation, as long as the bound $N/8$ is always observed.

References

1. M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1983. ACM.
2. Miklos Ajtai. Oblivious rams without cryptographic assumptions. In *STOC '10: Proceedings of the 42nd annual ACM symposium on Theory of computing*, 2010. To be published at STOC.
3. K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. ACM.
4. Paul Beame and Widad Machmouchi. Making RAMs Oblivious Requires Superlogarithmic Overhead. *Electronic Colloquium on Computational Complexity (ECCC)*, 10(104), 2010.
5. Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194, New York, NY, USA, 1987. ACM.
6. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.