Perfectly Secure Oblivious RAM Without Random Oracles

Preliminary version

Ivan Damgård Sigurd Meldgaard
Jesper Buus Nielsen
Department of Computer Science, Aarhus University

March 2, 2010

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 lover bound on the amount of randomness needed for information theoretically secure oblivious RAM.

1 Introduction

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

A trivial solution is to access all data every time one piece of data is needed. However, many algorithms are designed for being effective 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 [5] 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 [5] 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 pseudorandom 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. However, this is not possible: in this paper we show a lower bound on the number of random bits that an oblivious RAM simulator must use to hide the access pattern information theoretically. 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 multiparty computation. The advantage of the 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 multiparty computation setting.

Using this approach, we construct an oblivious RAM simulator where we can make an access in amortized time $O(\sqrt{N}\log(N))$ where N is the size of the memory provided, and next improve that solution ending up with only a polylogarithmic overhead per access.

Finally we prove a lover bound of essentially $\log N$ on the number of random bits an oblivious RAM must use for every read operation executed.

In recent concurrent and independent work [2] Ajtai also deals with oblivious RAM and unconditional security. His result appears to solve essentially the same problem as we do, but using a different technique that does not seem to lead to a zero-error solution.

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, so that when the results are recombined, the result will be as if the computation had been done on the original inputs.

  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. Using an oblivious RAM, we can hide the access pattern and this allows us, for instance, to do array indexing with secret indices in a straightforward way [1]. Using the standard approach of writing the desired computation as an arithmetic circuit, indexing with secret values is cumbersome and involves an overhead linear in the size of the array.

  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

A Random Access Machine of size n can be seen as an interactive functionality that, in round i, given input $(store, p_i, value_i)$ with $value_i \in F$, responds *ok*. And on input $(read, p_i)$, it responds $value_j$, where $j < i$ is the highest index, such that there have been issued a $(store, p_j, value_j)$ with $p_j = p_i$, and 0 if there has been no storing at p_i. We call a sequence

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

of *store*/*read* requests an access pattern. We can also simplify the interface, so it both retrieves the value, and saves another value at the same location. We call this a lookup.

A simulation of a RAM is a functionality with a bounded internal memory (constant size for example), that implements the interface of a RAM, using auxillary access to another RAM (the physical RAM).

We say that such a simulation securely implements an ideal oblivious RAM, if for any two access patterns to the ideal RAM, the respective access patterns that the simulation makes to the physical RAM are indistinguishable.

We assume the simulation has constant time access to operations $*$ and $+, -$ on its local values taking two inputs, and yielding the product/sum/difference. And also $<$, $=$, for comparison and equality test, taking two inputs x, y and outputting a bit; 1 if $x < y/x = y$ and 0 otherwise.

4 Oblivious sorting and shuffling

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

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, and so in asymptotically the same (expected) time, we can have a perfect solution. This issue is, in fact, the only source of error in our solution.

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

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

If each record contains its original index, we can use these fields to shuffle according to the inverse permutation by sorting according to this index.

5 The square root algorithm

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

Like the original square root algorithm of Goldreich in [4], the algorithm works by having a randomized dictionary structure that is used to connect an index to a piece of data. At the same time we store a linear cache of previously accessed elements, so we can make sure not to look at the same place twice, and we also use dummy elements to hide whether we access the same place twice. And we also amortize the time used for searching the increasingly long list by reshuffling everything for every $\sqrt{N}$ accesses.

But without a random oracle we cannot store a random permutation for free (by storing a key to the oracle), so we save the permutation using a binary tree with each level shuffled individually, so each branch of 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 first level is not shuffled.

5.1 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, a .used field that is used for reshuffling the data as described below.

Algorithm 5.1: DISPATCH($index, record$)

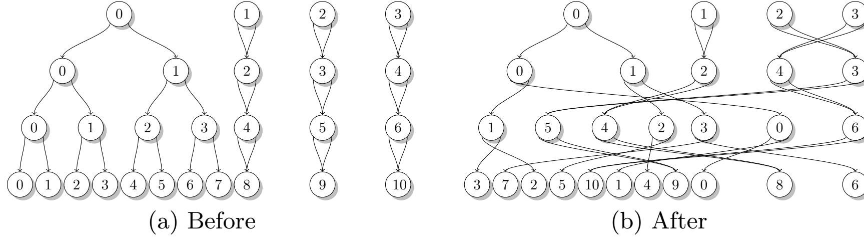
```
output: The left or right child of record, depending on index
if index < record.bound
  then return (record.left)
  else return (record.right)
```

Algorithm 5.2: LOOKUP($index$)

```
input: index
output: Value stored at index
if count ≥ √n
  then { UNMINGLE()
         SHUFFLE()
         count ← 0
  else count ← count + 1
next ← count
next_in_tree ← count
for level ← 0 to log2(N) − 1
  do { match ← False
       for i ← 0 to count − 1
         do { k ← cache[level, i]
              if k.index = next
                then { k_from_cache ← k
                       match ← True
                       k.index = ∞
                       cache[level, i] ← k        (1)
       if match
         then next ← next_from_tree
       k_from_tree ← physical[level, next]
       physical[level, next].used = True
       next_from_tree ← DISPATCH(index, k_from_tree)
       next ← DISPATCH(index, k_from_tree)
       if match
         then next ← DISPATCH(index, k_from_tree)
       cache[level, count] ← (next, UPDATE(k))
```

An invariant for the outer loop can be phrased:

1. *next* is the real index we are going to look for at level



(a) Before (b) After

Figure 1: Visualization of the memory layout of a tree storing 8 elements, before and after shuffling the tree. The edges indicate child-pointers.

2. *next_in_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 in line 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.

5.2 Obliviously shuffling a tree

We present an algorithm for shuffling a tree, it works by shuffling one level at a time, starting at the bottom of the tree, and ending with the root-level.

After shuffling a level, it does the reverse shuffle on a sequence $\{1 \ldots\}$, and copies these numbers into the pointer fields on the level above. 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 5.3: UNMINGLE( )

$a \leftarrow$ Filter out any physical[$\log_2(N)$] 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 5.4: SHUFFLE( )

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

5.3 Security

The transcript of a lookup, as seen by the adversary, always consist 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 $\sqrt{N}$ access, 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.

5.4 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 $O(\sqrt{N}\log^2(N))$ per lookup.

6 Polylogarithmic Solution

It is also possible to get a solution with an amortized overhead per access of $\log^{O(1)}(N)$. We introduce the notation $\tilde{O}$ which swallows polylogarithmic factors. In this notation we have a solution with overhead $\tilde{O}(1)$. The solution proceeds somewhat like the protocol [5] with polylogarithmic overhead. We will here only sketch the solution.

As in [5] the idea is to have $\log(N)$ levels of simulated RAMs. The simulated RAM at level 1 has size $\tilde{O}(N)$, and in general the simulated RAM at level ℓ has size $\tilde{O}(N/2^\ell)$. In [5] each level is a dictionary reminiscent of the one used in their square root solution, and in our solution each level is a shuffled tree, as used in our square root solution.

The main ideas behind the solution are as follows:

1. We start with a random shuffled tree at level 1 with all data in this tree and all other levels being empty.
2. When a path $(\nabla_1, \ldots, \nabla_m)$ in the tree of any level is touched, the nodes ∇_i are moved to the bottom level, i.e., level $\log(N)$, and the path is shuffled such that the nodes ∇_i are stored in random positions at the bottom level and such that ∇_i points to the new physical location of ∇_{i+1} at the bottom level.[2]

   The pointers from the ∇_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 points to untouched nodes in the trees at higher levels.
3. When the tree at level ℓ reaches size $\tilde{O}(N/2^\ell)$ the nodes at the lower levels $\log(N), \ldots, \ell$ are reshuffled such that all nodes are placed at new random positions and their intra-pointers are updated. Pointers to untouched nodes at higher levels are kept the same. The result is stored at level $\ell - 1$.
4. During the movement of nodes, either their moving to level $\log(N)$ after being touched or their being shuffled higher up the levels, it is always remembered where the root node of the tree original at level 1 is stored. Any search starts at this node. This is oblivious as any search starts at the root no matter the index being searched for.
5. If we were just to start each search at the original 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

[2] This is trivial to do with overhead $\tilde{O}(1)$ as the bottom level has size $\tilde{O}(1)$ and $m = O(\log(N)) = \tilde{O}(1)$.

pattern of how the path jumps between the levels would however leak information.[3] We therefore make such to touch each level once every time we follow one pointer. 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.

The above ensures that the access pattern is oblivious. 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. Moving the $\log(N)$ nodes that we looked up to level 1 and shuffling them can be done in $\tilde{O}(1)$, as discussed above, and the merging and shuffling of nodes at lower levels to higher levels amortizes to $\tilde{O}(1)$ per update, as levels of size s are shuffled only at every $s/\log(N)$ update, as each update pushed down $\log(N)$ nodes such that the total number of nodes in the lower levels grow by $\log(N)$ in each update.

7 Lower bound

An equivalent way to state the definition of a secure oblivious RAM simulation is that, for every program, the distribution of memory access 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.

In this section, we assume that the simulation is data-oblivious, i.e., the simulation never looks at the data it is asked to read or write (but it may of course look at, e.g., randomness it wrote itself in physical memory).

We will show that this implies a lower bound on the amount of randomness that the simulation must use. Consider the program that first writes random data to all N locations in RAM. It then executes k read operations from randomly chosen locations.

Let P be the random variable describing the choice of locations to read from. Clearly $H(P) = k \log N$. Let C 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 reads.

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 follows by elementary information theory that

$$H(K|C) \geq H(P) - H(P|C,K) = k \log N - H(P|C,K)$$

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, and let M_i represent the value of internal memory at the start of the i'th read operation. The plan is now to bound $H(P_i|C = c, K = k, M_i = m) := H(P_i|c,k,m)$ for arbitrary, fixed values c, k, m, use this to bound $H(P_i|C = c, K = k) = H(P_i|c,k)$ and finally use this to bound $H(P|C,K)$.

Note first that once we fix $K = k, M_i = m$, the choice of n locations to access depends only on P_i, since the simulation is data-oblivious. Hence, when we fix $C = c$, this will constrain the choice of P_i to be uniform over those values that cause the n locations specified in c to be accessed. Let w be this number of remaining choices for P_i. We then have that $H(P_i|c,k,m) = \log w$.

Let a be the number of bits in a memory location, and let $D_{c,k,m}$ represent the data written to the w locations in question, with the distribution implied by he choices of c, k, m. Let $D_{c,k}$ be the distribution of data written to those same w words, but now only given c, k.

[3] 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 touched 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 level 1. This would allow the adversary to distinguish.

$D_{c,k}$ is uniform over the 2^{aw} possible values, again because the simulation is data oblivious, so fixing $C = c, K = k$ does not constrain the data stored in the w locations in question.

However, we want to estimate how uncertain the simulator is about $D_{c,k,m}$, and must therefore take internal memory into account. To this end, we will assume in the following that all $P_j, j < i$ and all data written to locations other than the w in question are fixed to arbitrary values consistent with $C = c, K = k$, this will not affect the distribution of P_i. Once we do this, the value of M_i is a deterministic function f of $D_{c,k}$.

Let $pre(m)$ be the size of the preimage under f of m, then the probability that $M_i = m$ conditioned on $K = k, C = c$ is $p_m = pre(m)/2^{aw}$. We now have

$$H(D_{c,k,m} = \log pre(m) = \log p_m + aw$$

Let $R_{c,k,m}$ represent the data the data read from memory given the fixed access pattern c. Clearly, $H(R_{c,k,m}) \leq an$. But on the other hand $H(R_{c,k,m})$ must be larger than $H(D_{c,k,m})$, since otherwise the simulator does not have enough information to return a correct result of the read operation. More precisely, if the simulation always returns correct results, we can reconstruct the exact value of $D_{c,k,m}$ as a deterministic function of $R_{c,k,m}$ by letting the value of P_i run through all w possibilities and seeing what the simulation would return. Since applying a deterministic function can only decrease entropy, it must be that $H(R_{c,k,m}) \geq H(D_{c,k,m})$.

We therefore conclude that $na \geq \log p_m + aw$ or

$$w \leq n - \frac{\log p_m}{a}$$

Combining with the above, we get

$$H(P_i|c,k,m) = \log w \leq \log(n - \frac{\log p_m}{a})$$

By definition of conditional entropy, and writing u for the number of memory words we have room for in internal memory, we get

$$\begin{aligned}
H(P_i|c,k) &= \sum_m p_m H(P_i|c,k,m) \\
&\leq \sum_m p_m \log(n - \frac{\log p_m}{a}) \\
&\leq \log(\sum_m p_m(n - \frac{\log p_m}{a})) \\
&= \log(n + H(M_i|c,k)/a) \\
&\leq \log(n+u)
\end{aligned}$$

where the second inequality above is Jensen's inequality. By standard properties of entropy, we have

$$H(P|c,k) \leq \sum_i H(P_i|c,k) \leq k\log(n+u).$$

So we also have $H(P|K,C) \leq k\log(n+u)$ by definition of conditional entropy.

Combining all of the above we therefore have:

Theorem 7.1. *Suppose we are given perfectly secure oblivious RAM simulation of a memory of size N, accessing at most n locations in pysical RAM per read operation, and using internal memory of size at most u words. If such a simulation is data-oblivious, it must use at least $k\log(N/(n+u))$ random bits that are unknown to the adversary to execute k read operations.*

If the simulation is only statistically secure, this means that P and C are not independent, rather the distributions of C caused by different choices of P are statistically close. This means that the information overlap $I(C;P)$ is negligibly small as a function of some

security parameter. It is straightforward to see that if we drop the assumption that P, C are independent the first inequality above becomes

$$H(K) \geq k \log N - H(P|C, K) - I(P; C),$$

and since the rest of the argument for the theorem does not depend on C, P being independent, we see that the lower bound changes by only a negligible amount if the simulation is only statistically secure.

Data Non-Oblivous Simulation

The above argument 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 nevertheless believe that a lower bound holds even for non-oblivious simulation: Since C is independent of D and K is independent of D, it follows that $I(K; D|C) = I(C; D|K)$. So if $I(C; D|K)$ is large, then $H(K|C) \geq I(K; D|C)$ is large as well, and we are happy. On the other hand, if $I(C; D|K)$ is small, this means that C is almost independent of D, even when K is given. And this intuitively seems to mean that above argument which assumes full independence, should “almost” hold.

In any case, we do not see this issue as very important: it is clear that for our main application to multiparty computation, and probably in general as well, a data oblivious simulation is what we want.

8 Conclusion

We have shown how to transfer the results of Goldreich and Ostrovsky to the standard model, resulting in an algorithm for implementing a information theoretically secure oblivious RAM with a $\sqrt{N} \log^2(N)$ amortized overhead per access, without the need for a random oracle, and we then improved this to obtain a poly logarithmic overhead. Finally we have also shown a lower bound of essentially $\log(N)$ random bits per read operation based on information theoretic arguments.

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] O. 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.

[5] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.