A Simple ORAM

Kai-Min Chung[*] Rafael Pass[*]

April 29, 2013

Abstract

In this short note, we demonstrate a simple and practical ORAM that enjoys an extremely simple proof of security. Our construction is based on a recent ORAM due to Shi, Chan, Stefanov and Li [SCSL11], but with some crucial modifications, which significantly simply the analysis.

[*]Cornell University. {chung,rafael}@cs.cornell.edu

Chung is supported in part by NSF Award CCF-1214844 and Pass' Sloan Fellowship.
Pass is supported in part by a Alfred P. Sloan Fellowship, Microsoft New Faculty Fellowship, NSF Award CNS-1217821, NSF CAREER Award CCF-0746990, NSF Award CCF-1214844, AFOSR YIP Award FA9550-10-1-0093, and DARPA and AFRL under contract FA8750-11-2- 0211. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

1 Introduction

In this short note we consider constructions of *Oblivious RAM (ORAM)* [Gol87, GO96]. Roughly speaking, an ORAM enables executing a RAM program while hiding the access pattern to the memory. ORAM have several fundamental applications (see e.g. [GO96, OS97] for further discussion), but to date, all ORAMs require quite sophisticated and complicated analyses (which have lead to several flawed analyses; see e.g., [KLO12] for further discussion on this point).

Our goal here is to provide an ORAM with an extremely simple proof of security. Additionally, our solution does not rely on any cryptographic hardness assumptions (or random oracles).

Theorem 1 (Informally stated)**.** *There exists an ORAM with* $\operatorname{poly}\log(n)$ *worst-case computational overhead and* $\omega(\log n)$ *memory overhead, where* n *is the memory size.*[1]

Our construction is based on on a recent elegant ORAM construction due to Shi, Chan, Stefanov and Li [SCSL11], but with some crucial modifications, which significantly simply the analysis (and in our eyes also make the construction conceptually simpler).

2 Defining ORAM

A Random Access Machine (RAM) with memory size n consists of a CPU with a small number of registers (e.g., constant or $\operatorname{poly}\log(n)$) that each can store a string of length $\log n$ (called a word) and an "external" memory of size n. To simplify notation, a word is either $\perp$ or a $\log n$ bit string.

The CPU executes a program Π (given n and some input x) that can access the memory by a $Read(r)$ and $Write(r, val)$ operations where $r \in [n]$ is an index to a memory location, and val is a word (of size $\log n$). The sequence of memory cell accesses by such read and write operations is referred to as the *memory access pattern* of $\Pi(n, x)$ and is denoted $\tilde{\Pi}(n, x)$. (The CPU may also execute "standard" operations on the registers, any may generate outputs).

Let us turn to defining an *Oblivous RAM Compiler.* This notion was first defined by Goldreich [Gol87] and Goldreich and Ostrovksy [GO96]. We here provide a more succinct variant of their definition.

Definition 1. *A polynomial-time algorithm* C *is an* Oblivious RAM (ORAM) compiler *with computational overhead* $c(\cdot)$ *and memory overhead* $m(\cdot)$*, if* C *given* $n \in N$ *and a deterministic RAM program* Π *with memory-size* n *outputs a program* Π' *with memory-size* $m(n) \cdot n$ *such that for any input* x*, the running-time of* $\Pi'(n, x)$ *is bounded by* $c(n) \cdot T$ *where* T *is the running-time of* $\Pi(n, x)$*, and there exists a negligible function* μ *such that the following properties hold:*

- **Correctness:** *For any* $n \in N$ *and any string* $x \in \{0, 1\}^*$*, with probability at least* $1 - \mu(n)$*,* $\Pi(n, x) = \Pi'(n, x)$*.*
- **Obliviousness:** *For any two programs* Π_1, Π_2*, any* $n \in N$ *and any two inputs* $x_1, x_2 \in \{0, 1\}^*$ *if* $|\tilde{\Pi}_1(n, x_1)| = |\tilde{\Pi}_2(n, x_2)|$*, then* $\tilde{\Pi}'_1(n, x_1)$ *is* μ*-close to* $\tilde{\Pi}'_2(n, x_2)$ *in statistical distance, where* $\Pi'_1 = C(n, \Pi_1)$ *and* $\Pi'_2 = C(n, \Pi_2)$*.*

Note that the above definition (just as the definition of [GO96]) only requires an oblivious compilation of *deterministic* programs Π. This is without loss of generality: we can always view a randomized program as a deterministic one that receives random coins as part of its input.

[1] If using a CPU with $\operatorname{poly}\log n$ registers, the computational overhead is $\omega(\log^3 n)$; if using a CPU with a constant number of registers, the computational overhead is $\omega(\log^4 n)$.

3 The ORAM Construction

Our construction closely follows the general approach of [SCSL11]. We start by providing a solution where the CPU needs to have a "huge" $n/\alpha + \text{poly}\log n$ number of registers, where $\alpha > 1$ is any constant. This solution can then be applied recursively to bring down the number of registers to be polylogarithmic, by only blowing up the computational overhead by a factor $\log n$. (Finally, as we note in Remark 1, at the cost of another factor $\omega(\log n)$ in computational overhead, the number of registers can also be brought down to a constant, but in our opinion, the model of having a polylogarthmic number of registers corresponds better to practice.)[2]

The basic construction: ORAM with $O(n)$ registers Our compiler C on input $n \in N$ and a program Π with memory size n outputs a program Π' that is identical to Π but each $read(r)$ or $write(r, val)$ operation, is replaced by a sequence of operations defined by subroutines $Oread(r)$ and $Owrite(r, val)$ to be specified shortly. Π' has the same registers as Π and additionally has n/α registers used to store a *position map* Pos plus a polylogarithmic number of additional *work* registers used by $Oread$ and $Owrite$. In its external memory, Π' will maintain a complete binary tree Γ of depth $d = \log(n/\alpha)$; we index nodes in the tree by a binary string of length at most d, where the root is indexed by the empty string λ, and each node indexed by γ has left and right children indexed $\gamma 0$ and $\gamma 1$, respectively. Each memory cell r will be associated with a random leaf pos in the tree, specified by the position map Pos; as we shall see shortly, the memory cell r will be stored at one of the nodes on the path from the root λ to the leaf pos. To ensure that the position map is smaller than the memory size, we assign a *block* of α consecutive memory cells to the same leaf; thus memory cell r corresponding to block $b = \lfloor r/\alpha \rfloor$ will be associated with leaf $Pos(b)$. See Figure 1 for an illustration of position map and the ORAM tree.

Each node in the tree is associated with a *bucket* which stores (at most) K tuples (b, pos, v) where v is the content of block b and pos is the leaf associated with the block b; $K \in \omega(\log n) \cap \text{poly}\log(n)$ is a parameter that will determine the security of the ORAM (thus each bucket stores $K(\alpha + 2)$ words.) We assume that all registers and memory cells are initialized with a special symbol $\perp$.

We next specify $Oread(r)$, which proceeds in the following steps.

Fetch: Let $b = \lfloor r/\alpha \rfloor$ be the block containing memory cell r, and let $i = r \mod \alpha$ be the r's component in the block b. We first look up the position of the block b using the position map: $pos = Pos(b)$; if $Pos(b) = \perp$, let $pos \leftarrow [n/\alpha]$ to be a uniformly random leaf.

Next, we traverse the tree from the roof to the leaf pos, making exactly one read and one write operation for every memory cell associated with the nodes along the path. More precisely, we read the content once, and then we either write it back (unchanged), or we simply "erase it" (writing $\perp$) so as to implement the following task: search for a tuple of the form (b, pos, v) in any of the nodes during the traversal; if such a tuple is found, remove it, and otherwise let $v = \perp$. Finally return the ith component of v.

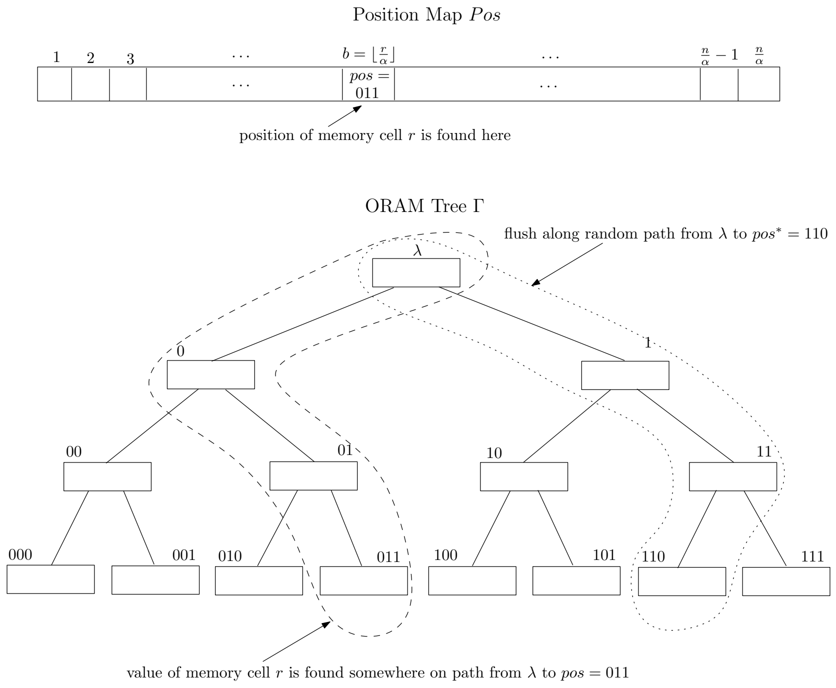
Update Position Map: Pick a uniformly random leaf $pos' \leftarrow [n/\alpha]$ and let $Pos(b) = pos'$.

Put Back: Add the tuple (b, pos', v) to the root λ of the tree. If there is not enough space left in the bucket, abort outputting **overflow**.

Flush: [3] Pick a uniformly random leaf $pos^* \leftarrow [n/\alpha]$ and traverse the tree from the roof to the leaf pos^*, making exactly one read and one write operation for every memory cell associated

[2] In particular if we view the cache of a CPU as its internal registers.

[3] We mention that this is the main step where our construction is different from Shi et al [SCSL11].



Position Map Pos
1
2
3
...
$b = \lfloor \frac{r}{\alpha} \rfloor$
...
$\frac{n}{\alpha} - 1$
$\frac{n}{\alpha}$
...
$pos =$ 011
...
position of memory cell r is found here
ORAM Tree Γ
flush along random path from λ to $pos^* = 110$
λ
0
1
00
01
10
11
000
001
010
011
100
101
110
111
value of memory cell r is found somewhere on path from λ to $pos = 011$

Figure 1: Illustration of the basic ORAM construction

with the nodes along the path so as to implement the following task: "push down" each tuple (b'', pos'', v'') read in the nodes traversed as far as possible along the path to pos^* while ensuring that the tuple is still on the path to its associated leaf pos'' (that is, the tuple ends up in the node $\gamma =$ longest common prefix of pos'' and pos^*.) (Note that this operation can be done trivially as long as the CPU has sufficiently many work registers to load two whole buckets into memory; since the bucket size is polylogarithmic, this is possible.) If at any point some bucket is about to overflow, abort outputting **overflow**.

$Owrite(r, val)$ proceeds in identically the same steps as $Oread(r)$, except that in the "Put Back" steps, we add the tuple (b, pos', v') where v' is the string v but the ith component is set to val (instead of adding the tuple (b, pos', v) as in $Oread$). (Note that, just as $Oread$, $Owrite$ also outputs the original memory content of the memory cell r; this feature will be useful in the "full-fledged" construction.)

The following observation is central to the construction of our ORAM (and an appropriate analog of it was central already to the construction of [SCSL11]):

> **Key observation:** *Each Oread and Owrite operation traverses the the tree along two randomly chosen paths,* independent *of the history of operations so far.*

The key observation follows from the facts that (1) just as in the scheme of [SCSL11], each position in the position map is used exactly once in a traversal (and before this traversal, no information about the position is used in determining what nodes to traverse), and (2) the flushing, by definition, traverses a random path, independent of the history.

The full-fledged construction: ORAM with polylog registers The full-fledged construction of our ORAM proceeds just as the above, except that instead of storing the position map in registers in the CPU, we recursively store them in another ORAM (which only needs to operate on n/α memory cells, but still using a bucket that store K tuples). Recall that each invokation of *Oread* and *Owrite* requires reading one position in the position map and updating its value to a random leaf; that is, we need to perform a *single* recursive *Owrite* call (recall that *Owrite* updates the value in a memory cell, and returns the old value) to emulate the position map.

At the base case of the recursion, when position map is of constant size, we use the basic ORAM construction which simply stores the position map in the registers.

Comparison with the construction of Shi et al. As mentioned, our ORAM is closely related to the ORAM of Shi et al [SCSL11]. The main difference is that instead of having a flush operation, the ORAM of [SCSL11] performs an "eviction procedure" that is somewhat more elaborate than the flushing step; more important, analyzing the overflow probability when using their eviction procedue is non-trivial (involving analyzing the stationary distribution of a non-trivial Markov chain; as far as we know, a full analysis has not yet been made public.) In contrast, by using our flushing operation, the analysis becomes elementary.

3.1 Analysis of The ORAM

Before analysing the ORAM, let us first describe a simple "dart game" which abstracts out the central combinatorial reason for why there are no overflows in the ORAM.

A dart game: You have an unbounded number of white and black darts. In each round of the game, you first throw a black dart, and then a white dart; each dart independently hits the bullseye with probability p. You continue the game until at least K darts have hit the bullseye. You "win" if none of darts that hit the bullseye are white. What is the winning probability?

Note that for every winning dart sequence s, there exists $2^K - 1$ distinct other "loosing" sequences s' (where any non-empty subset of black darts hitting the bullseye are replaced with white darts), each of which happen with identically the same probability as the sequence s; additionally, every two distinct winning sequence s_1, s_2 yield disjoint loosing sequences. It follows that the winning probability is upper-bounded by 2^{-K}.

Let us now return to the ORAM problem. Given any program Π, let $\Pi'(n, x) = C(n, \Pi)(n, x)$.

Claim 1. *There exists negligible function μ such that for any deterministic program Π, any n and any input x, the probability that $\Pi'(n, x)$ outputs* $\mathsf{overflow}$ *is bounded by $\mu(n)$.*

Proof. Let us start by analyzing the basic construction. Let T denote the number of memory accesses in the execution of $\Pi(n, x)$. Consider any *internal node* (i.e., a node that is not a leaf) γ in the tree. For the bucket of node γ to overflow, there must be K tuples in it; by constructions each such tuple is of the form $(\cdot, \gamma||\cdot, \cdot)$ such that γ is a prefix of *pos*. There thus must exists some bit j such that the bucket contains at least $K/2$ tuples of the form $(\cdot, \gamma||j||\cdot, \cdot)$ (tuples with $j = 0/1$ belong to leave in left/right sub-trees, respectively).

Note that every time that we make a "flush" along a path associated with a leaf of the form $\gamma||j||\cdot$, we make sure that there are no tuples of the form $(\cdot, \gamma||j||\cdot, \cdot)$ on the path from the roof to node γ. Thus, in order for there to be $K/2$ tuples of the form $(b, \gamma||j||\delta, v)$ in bucket γ, we must have assigned $K/2$ leafs of the form $\gamma||j||\cdot$ to memory cells, without a single time having performed a flush associated with a leaf of the form $\gamma||j||\cdot$. Note that the probability of assigning a memory cell to a leaf of the form $\gamma||j||\cdot$ is identical to the probability of performing a flushing associated with a leaf of the form $\gamma||j||\cdot$; let's call this probability p. Thus, the probability of overflow in γ is upper-bounded by the probability of winning (at least once) in a sequence of at most T dart games (black darts hitting bullseye correspond to assigning a memory cell to a leaf of the form $\gamma||j||\cdot$, and white darts hitting bullseye correspond to performing a flushing associated with a leaf of the form $\gamma||j||\cdot$); by the union bound this probability is upper-bounded by $T2^{-K/2}$. Since there are $(n/\alpha)-1$ internal nodes in the tree, by another application of the union bound, it follows that the probability that there is an overflow in any of the internal node is bounded by $2^{-K/2}\cdot(n/\alpha)\cdot T$.

We turn to showing that the probability of overflow in any of the leaf nodes is small. Consider any leaf node γ and some time t. For there to be an overflow in γ at time t, there must be $K+1$ out of n/α elements in the position map that map to γ. Recall that all positions in the position map are uniformly and independently selected; thus, the expected number of elements mapping to γ is $\mu=1$ and by a standard multiplicative version of Chernoff bound, the probability that $K+1$ elements are mapped to γ is upper bounded by

$$\left(\frac{e^K}{(K+1)^{K+1}}\right)^{\mu}\leq 2^{-K/2}.$$[4]

It follows by an union bound over the number of leaf nodes and the total number of time steps that the probability of overflow in any of the leaf nodes throughout the execution is at most $2^{-K/2}\cdot(n/\alpha)\cdot T$.

By a final union bound, we have that the probability of *any* node ever overflowing is bounded by $2^{-(K/2)+1}\cdot(n/\alpha)\cdot T$

To analyze the full-fledged construction, we simply apply the union bound to the failure probabilities of the $\log_\alpha n$ different ORAM trees (due to the recursive calls). The final upper bound on the overflow probability is thus $2^{-(K/2)+1}\cdot(n/\alpha)\cdot T\cdot\log_\alpha n$, which is negligible as long as $K\in\omega(\log n)$. □

Correctness of the ORAM By construction it follows that for any deterministic program Π, any n and any input x, as long as $\Pi'(n,x)$ does not output $\mathsf{overflow}$, its output is identical to the output of $\Pi(n,x)$; by Claim 1, it follows that C satisfies the correctness condition of an ORAM compiler.

Obliviousness Consider two deterministic programs Π_1 and Π_2 and inputs x_1,x_2 such that $|\tilde{\Pi}_1(x_1)|=|\tilde{\Pi}_2(x_2)|$. It directly follows the key observation that conditioned on $\Pi_1'(n,x_1)$ and $\Pi_2'(n,x_2)$ not outputting $\mathsf{overflow}$, their memory access patterns are identically distributed. By Claim 1 and a union bound, we have that the probability that either $\Pi_1'(n,x_1)$ or $\Pi_2'(n,x_2)$ output $\mathsf{overflow}$ is negligible; we conclude that C satisfies the obliviousness condition.

Computational and Memory Overhead Let us first consider the overhead of the basic construction. Recall that the compiled program Π' uses a tree with $(2n/\alpha)-1$ nodes, and each node stores a bucket of $K(\alpha+2)$ words; thus the memory overhead is $O(K)$. For the computational overhead, note that Π' is identical to Π except that each memory access operation is replaced by an execution of *Oread* or *Owrite*. Recall that each execution of *Oread* or *Owrite* traverses the

[4]We use the following version of the Chernoff bound: Let $X_1,\ldots,X_n$ be independent $[0,1]$-valued random variables. Let $X=\sum_i X_i$ and $\mu=\mathbb{E}[X]$. For every $\delta>0$,

$$\Pr\left[X\geq(1+\delta)\mu\right]\leq\left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\mu}.$$

ORAM tree twice and each traversal read and write each bucket of the traversed nodes exactly once. The computational overhead is thus $O(K \log(n/\alpha))$.

We now consider the overhead of the full-fledged construction. Since each recursive call reduces the memory size by a constant factor of $\alpha > 1$, the memory overhead is $O(K) + O(K/\alpha) + O(K/\alpha^2) + \cdots + O(1) = O(K)$. For the computational overhead, recall that each memory access operation requires performing one "recursive" *Owrite* operation for accessing (and updating) the "outsourced" position maps. Since there are $\log_\alpha n$ recursive levels, the computational overhead is at most $O(K \log(n/\alpha) \log_\alpha n)$.

Remark 1 (ORAM with a constant number of registers). The only reason our ORAM requires using $O(K)$ registers is to implement the "flush" operation while only reading and writing each node traversed *once*. More specifically, to "push down" tuples from one bucket (i.e., node on the tree) to the next, we require loading both buckets into memory. But we can also implement this "push down" operation using a constant number of registers, at the cost of an additional $O(K)$-factor in computational overhead. More precisely, to "push down" tuples from a node γ to its child $\gamma||b$, we simply move down tuples in γ *one-by-one*. To ensure obliviousness, we need to go over every potential tuple in γ (that is, $O(K)$ tuples) and individually push them down, and for each such individidual push-down, we need to scan through the whole receiving bucket $\gamma||b$ (that is, we need to scan through $O(K)$ tuples); thus in total $O(K^2)$ memory accesses are needed, as opposed the $O(K)$ accesses used if we can read two whole buckets into registers.

References

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996. 1

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987. 1

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *SODA*, pages 143–156, 2012. 1

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997. 1

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In *ASIACRYPT*, pages 197–214, 2011. 1, 2, 3, 4