Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead

Kai-Min Chung[1], Zhenming Liu[2], and Rafael Pass[⋆3]

[1] Institute of Information Science, Academia Sinica, Taiwan.
kmchung@iis.sinica.edu.tw
[2] Department of Computer Science, Princeton University, Princeton, NJ, USA.
lzhenming@post.harvard.edu
[3] Department of Computer Science, Cornell NYC Tech, Ithaca, NY, USA.
rafael@cs.cornell.edu

Abstract. We demonstrate a simple, statistically secure, ORAM with computational overhead $\tilde{O}(\log^2 n)$; previous ORAM protocols achieve only computational security (under computational assumptions) or require $\tilde{\Omega}(\log^3 n)$ overheard. An additional benefit of our ORAM is its conceptual simplicity, which makes it easy to implement in both software and (commercially available) hardware.

Our construction is based on recent ORAM constructions due to Shi, Chan, Stefanov, and Li (Asiacrypt 2011) and Stefanov and Shi (ArXiv 2012), but with some crucial modifications in the algorithm that simplifies the ORAM and enable our analysis. A central component in our analysis is reducing the analysis of our algorithm to a "supermarket" problem; of independent interest (and of importance to our analysis,) we provide an upper bound on the rate of "upset" customers in the "supermarket" problem.

1 Introduction

In this paper we consider constructions of *Oblivious RAM (ORAM)* [10,11]. 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. [11,24] for further discussion). Since the seminal works for Goldreich [10] and Goldreich and Ostrovksy [11], constructions of ORAM have been extensively studied. See, for example, [32,33,1,25,12,6,27,2,13,29,15] and references therein. While the original constructions only enjoyed "computational security" (under the the assumption that one-way functions exists) and required a computational overhead of $\tilde{O}(\log^3 n)$, more recent works have overcome both of these barriers, but only individually. State of the art ORAMs satisfy either of the following:

⋆ Pass is supported in part by a Alfred P. Sloan Fellowship, Microsoft New Faculty Fellowship, NSF CAREER Award CCF-0746990, 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.

– An overhead of $\tilde{O}(\log^2 n)$[4], but only satisfies computational security, assuming the existence of one-way functions. [25,12,15]
– Statistical security, but have an overhead of $O(\log^3 n)$. [1,6,27,8,5].

A natural question is whether both of these barriers can be simultaneously overcome; namely, does there exists a statistically secure ORAM with only $\tilde{O}(\log^2 n)$ overhead? In this work we answer this question in the affirmative, demonstrating the existence of such an ORAM.

Theorem 1. *There exists a statistically-secure ORAM with* $\tilde{O}(\log^2(n))$ *worst-case computational overhead, constant memory overhead, and CPU cache size* $\operatorname{poly}\log(n)$*, where* n *is the memory size.*

An additional benefit of our ORAM is its conceptual simplicity, which makes it easy to implement in both software and (commercially available) hardware. (A software implementation is available from the authors upon request.)

Our ORAM Construction A conceptual breakthrough in the construction of ORAMs appeared in the recent work of Shi, Chan, Stefanov, and Li [27]. This work demonstrated a statistically secure ORAM with overhead $O(\log^3 n)$ using a new "tree-based" construction framework, which admits significantly simpler (and thus easier to implemented) ORAM constructions (see also [8,5] for instantiations of this framework which additionally enjoys an extremely simple proof of security).

On a high-level, each memory cell r accessed by the original RAM will be associated with a random leaf pos in a binary tree; the position is specified by a so-called "position map" Pos. Each node in the tree consists of a "bucket" which stores up to ℓ elements. The content of memory cell r will be found inside one of the buckets along the path from the root to the leaf pos; originally, it is put into the root, and later on, the content gets "pushed-down" through an eviction procedure—for instance, in the ORAM of [5] (upon which we rely), the eviction procedure consists of "flushing" down memory contents along a random path, while ensuring that each memory cell is still found on its appropriate path from the root to its assigned leaf. (Furthermore, each time the content of a memory cell is accessed, the content is removed from the tree, the memory cell is assigned to a new random leaf, and the content is put back into the root).

In the work of [27] and its follow-ups [8,5], for the analysis to go through, the bucket size ℓ is required to be $\omega(\log n)$. Stefanov and Shi [28] recently provided a different instantiation of this framework which only uses *constant size* buckets, but instead relies on a *single* $\operatorname{poly}\log n$ size "stash" into which potential "overflows" (of the buckets in the tree) are put;[5] Stefanov and Shi conjectured (but did not prove) security of such a construction (when appropriately evicting elements from the "stash" along the path traversed to access some memory cell).[6]

In this work, we follow the above-mentioned approaches, but with the following high-level modifications:

[4] The best protocol achieves $O(\log^2 n/ \log\log n)$.
[5] We mention that the idea of using "stash" also appeared in the works [12,13,15,17].
[6] Although different, the "flush" mechanism in [5] is inspired by this eviction method.

- We consider a binary tree where the bucket size of all *internal* buckets is $O(\log \log n)$, but all the leaf nodes still have bucket size $\omega(\log n)$.
- As in [28], we use a "stash" to store potential "overflows" from the bucket. In our ORAM we refer to this as a "queue" as the main operation we require from it is to insert and "pop" elements (as we explain shortly, we additionally need to be able to find and remove any particular element from the queue; this can be easily achieved using a standard hash table). Additionally, instead of inserting memory cells directly into the tree, we insert them into the queue. When searching for a memory cell, we first check whether the memory cell is found in the queue (in which case it gets removed), and if not, we search for the memory cell in the binary tree along the path from the root to the position dictated by the position map.
- Rather than just "flushing" once (as in [5]), we repeat the following procedure "pop and random flush" procedure twice.
  - We "pop" an element from the queue into the root.
  - Next, we flush according to a *geometrically distributed* random variable with expectation 2.[7]

We demonstrate that such an ORAM construction is both (statistically) secure, and only has $\tilde{\Omega}(\log^2 n)$ overhead.

Our Analysis The key element in our analysis is reducing the security of our ORAM to a "supermarket" problem. Supermarket problems were introduced by Mitzenmacher [20] and have seen been well-studied (see e.g., [20,31,23,26,21]). We here consider a simple version of a supermarket problem, but ask a new question: what is the rate of "upset" customers in a supermarket problem: There are D cashiers in the supermarket, all of which have empty queues in the beginning of the day. At each time step t: with probability $\alpha < 1/2$ a new customer arrives and chooses a random cashier[8] (and puts himself in that cashiers queue); otherwise (i.e., with probability $1-\alpha$) a random cashier is chosen that "serves" the first customer in its queue (and the queue size is reduced by one). We say that a customer is *upset* if he chooses a queue whose size exceeds some bound φ. What is the rate of upset customers?[9]

We provide an upper bound on the rate of upset customers relying on Chernoff bounds for Markov chains [9,14,16,3]—more specifically, we develop a variant of traditional Chernoff bounds for Markov chains which apply also with "resets" (where at each step, with some small probability, the distribution is reset to the stationary distribution of the Markov chain), which may be of independent interest, and show how such a Chernoff bound can be used in a rather straight-forward way to provide a bound on the number of upset customers.

[7] Looking forward, our actual flush is a little bit different than the one in [5] in that we only pull down a *single* element between any two consecutive nodes along the path, whereas in [5] *all* elements that can be pulled down get flushed down.

[8] Typically, in supermarket problems the customer chooses d random cashiers and picks the one with the smallest queue; we here focus on the simple case when $d = 1$.

[9] Although we here consider a discrete-time version of the supermarket problem (since this is the most relevant for our application), as we remark in Remark 1, our results apply also to the more commonly studied continuous-time setting.

Intuitively, to reduce the security of our ORAM to the above-mentioned supermarket problem, each cashier corresponds to a bucket on some particular level k in the tree, and the bound φ corresponds to the bucket size, customers correspond to elements being placed in the buckets, and upset customers overflows. Note that for this translation to work it is important that the number of flushes in our ORAM is geometrically distributed—this ensures that we can view the sequence of operations (i.e., "flushes" that decrease bucket sizes, and "pops" that increase bucket sizes) as independently distributed as in the supermarket problem.

Independent Work In a very recent independent work, Stefanov, van Dijk, Shi, Fletcher, Ren, Yu, and Devadas [30] prove security of the conjectured Path ORAM of [28]. This yields a ORAM with overhead $O(\log^2 n)$, whereas our ORAM has overhead $O(\log^2 n \log\log n)$). On the other hand, the data structure required to implement our queue is simpler than the one needed to implement the "stash" in the Path ORAM construction. More precisely, we simply need a standard queue and a standard hash table (both of which can be implemented using commodity hardware), whereas the "stash" in [28,30,18] requires using a data structure that additionally supports sorting, or "range queries" (thus a binary search tree is needed), which may make implementations less straightforward. We leave a more complete exploration of the benefits of these two independent approaches for future work.

In another concurrent work, Gentry, Goldman, Halevi, Jutla, Raykova, and Wichs optimize the ORAM of [27]. In particular, they improve the memory overhead from $O(\log n)$ to constant, but the time overhead remains $\tilde{O}(\log^3 n)$. We rely on their idea to achieve constant memory overhead.

2 Preliminaries

A Random Access Machine (RAM) with memory size n consists of a CPU with a small size cache (e.g., can store a constant or $\text{poly}\log(n)$ number of words) 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 [10] and Goldreich and Ostrovksy [11]. We recall a more succinct variant of their definition due to [5].

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 [11]) 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.

3 Our ORAM and Its Efficiency

This section presents the construction of our ORAM, followed by an analysis of its efficiency.

3.1 The algorithm

Our ORAM data structure serves as a "big" memory table of size n and exposes the following two interfaces.
– READ(r): the algorithm returns the value of memory cell $r \in [n]$.
– WRITE(r,v): the algorithm writes value v to memory cell r, and returns the original value of r.

We start by assuming that the ORAM is executed on a CPU with cache size is $2n/\alpha + o(n)$ (in words) for a suitably large constant α (the reader may imagine $\alpha = 16$). Following the framework in [27], we can then reduce the cache size to $O(\text{poly}\log n)$ by recursively applying the ORAM construction; we provide further details on this transformation at the end of the section.

In what follows, we group each consecutive α memory cells in the RAM into a *block* and will thus have n/α blocks in total. We also index the blocks in the natural way, *i.e.* the block that contains the first α memory cells in the table has index 0 and in general the i-th block contains memory cells with addresses from αi to $\alpha(i+1)-1$.

Our algorithm will always operate at the block level, *i.e.* memory cells in the same block will always be read/written together. In addition to the content of its α memory cells, each block is associated with two extra pieces of information. First, it stores the index i of the block. Second, it stores a "position" p that specify its storage "destination" in the external memory, which we elaborate upon in the forthcoming paragraphs. In other words, a block is of the form (i, p, val), where val is the content of its α memory cells.

Our ORAM construction relies on the following three main components.
1. **A full binary tree at the in the external memory** that serves as the primary media to store the data.
2. **A position map in the internal cache** that helps us to search for items in the binary tree.
3. **A queue in the internal cache** that is the secondary venue to store the data.

We now walk through each of the building blocks in details.

The full binary tree Tr. The depth of this full binary tree is set to be the smallest d so that the number of leaves $L = 2^d$ is at least $2(n/\alpha)/(\log n \log\log n)$ (*i.e.*, $L/2 < 2(n/\alpha)/(\log n \log\log n) \leq L$). (In [27,5] the number of leaves was set to n/α; here, we instead follow [8] and make the tree slightly smaller—this makes the memory overhead constant.) We index nodes in the tree by a binary strings 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 node is associated with a *bucket*. A bucket in an internal node can store up to ℓ blocks, and a bucket in a leaf can store up to ℓ' blocks, where ℓ and ℓ' are parameters to be determined later. The tree shall support the following two atomic operations:

– $\textsc{Read}(\text{Node: } v)$: the tree will return all the blocks in the bucket associated with v to the cache.
– $\textsc{Write}(\text{Node: } v, \text{Blocks: } \boldsymbol{b})$: the input is a node v and an array of blocks $\boldsymbol{b}$ (that will fit into the bucket in node v). This operation will replace the bucket in the node v by $\boldsymbol{b}$.

The position map P. This data structure is an array that maps the indices of the blocks to leaves in the full binary tree. Specifically, it supports the following atomic operations:

– $\textsc{Read}(i)$: this function returns the position $P[i] \in [L]$ that corresponds to the block with index $i \in [n/\alpha]$.
– $\textsc{Write}(i, p)$: this function writes the position p to $P[i]$.

We assume that the position map is initialized with value $\perp$.

The queue Q. This data structure stores a queue of blocks with maximum size $q_{\max}$, a parameter to be determined later, and supports the following three atomic operations:

– $\textsc{Insert}(\text{Block } b)$: insert a block b into the queue.
– $\textsc{PopFront}()$: the first block in the queue is popped and returned.
– $\textsc{Find}(\text{int: } i, \text{word: } p)$: if there is a block b with index i and position p stored in the queue, then $\textsc{Find}$ returns b and deletes it from the queue; otherwise, it returns $\perp$.

Note that in addition to the usual $\textsc{Insert}$ and $\textsc{PopFront}$ operations, we also require the queue to support a $\textsc{Find}$ operation that finds a given block, returns and deletes it from the queue. This operation can be supported using a standard hash table in conjunction with the queue. We mention that all three operations can be implemented in time less than $O(\log n \log\log n)$, and discuss the implementation details in Appendix A.

Our Construction. We now are ready to describe our ORAM construction, which relies the above atomic operations. Here, we shall focus on the read operation. The algorithm for the write operation is analogous.

For two nodes u and v in Tr, we use $\mathrm{path}(u, v)$ to denote the (unique) path connecting u and v. Throughout the life cycle of our algorithm we maintain the following *block-path* invariance.

Block-path Invariance: *For any index $i \in [n/\alpha]$, either $P[i] = \perp$ and in this case both Tr and the queue do not contain any block with index i, or there exists a unique block b with index i that is located either in the queue, or in the bucket of one of the nodes on $\mathrm{path}(\lambda, P[i])$ in Tr*

We proceed to describe our $\textsc{Read}(r)$ algorithm. At a high-level, $\textsc{Read}(r)$ consists of two sub-routines $\textsc{Fetch}()$ and $\textsc{Dequeue}()$. $\textsc{Read}(r)$ executes $\textsc{Fetch}()$ and $\textsc{Dequeue}()$ once in order. Additionally, at the end of every $\log n$ invocations of $\textsc{Read}(r)$, *one extra* $\textsc{Dequeue}()$ is executed. Roughly, $\textsc{Fetch}()$ fetches the block b that contains the memory cell r from either $\mathrm{path}(\lambda, P[\lfloor r/\alpha \rfloor])$ in Tr or in Q, then returns the value of memory cell r, and finally inserts the block b to the queue Q. On the other hand, $\textsc{Dequeue}()$ pops one block b from Q, inserts b to the root λ of Tr (provided there is a room), and performs a *random* number of "$\textsc{Flush}$" actions that gradually moves blocks in Tr down to the leaves.

Fetch: Let $i = \lfloor r/\alpha \rfloor$ be the index of the block b that contains the r-th memory cell, and $p = P[i]$ be the current position of b. If $P[i] = \perp$ (which means that the block is not initialized yet), let $P[i] \leftarrow [L]$ be a uniformly random leaf, create a block $b = (i, P[i], \perp)$, and insert b to the queue Q. Otherwise, $\textsc{Fetch}$ performs the following actions in order.

Fetch from tree Tr **and queue** Q**:** Search the block b with index i along $\mathrm{path}(\lambda, p)$ in Tr by reading all buckets in $\mathrm{path}(\lambda, p)$ once and writing them back. Also, search the block b with index i and position p in the queue Q by invoking $\textsc{Find}(i, p)$. By the block-path invariance, we must find the block b.

Update position map P**.** Let $P[i] \leftarrow [L]$ be a uniformly random leaf, and update the position $p = P[i]$ of b.

Insert to queue Q**:** Insert the block b to Q.

Dequeue: This sub-routine consists of two actions $\textsc{Put-Back}()$ and $\textsc{Flush}()$. It starts by executing $\textsc{Put-Back}()$ once, and then performs a *random* number of $\textsc{Flush}()$es as follows: Let $C \in \{0, 1\}$ be a biased coin with $\Pr[C = 1] = 2/3$. It samples C, and if the outcome is 1, then it continues to perform one $\textsc{Flush}()$ and sample another independent copy of C, until the outcome is 0. (In other words, the number of $\textsc{Flush}()$ is a geometric random variable with parameter $2/3$.)

Put-Back: This action moves a block from the queue, if any, to the root of Tr. Specifically, we first invoke a $\textsc{PopFront}()$. If $\textsc{PopFront}()$ returns a block b then add b to λ .

Flush : This procedure selects a random path (namely, the path connecting the root to a random leaf $p^* \leftarrow \{0, 1\}^d$) on the tree and tries to move the blocks along the path down subject to the condition that the block always finds themselves on the appropriate path from the root to their assigned leaf node (see the block-path invariance condition). Let $p_0 (= \lambda) p_1 ... p_d$ be the nodes along $\mathrm{path}(\lambda, p^*)$. We traverse the path while carrying out the following operations for each node p_i we visit: in node p_i, find the block that can be "pulled-down" as far as possible along the path $\mathrm{path}(\lambda, p^*)$ (subject to the block-path invariance condition), and pull it down to p_{i+1}. For $i < d$, if there exists some $\eta \in \{0, 1\}$ such that p_i contains more than $\ell/2$ blocks that are assigned to leafs of the form $p_i||\eta||\cdot$ (see also Figure 1 in Appendix),[10] then select an arbitrary such block b, remove it from the bucket p_i and invoke an $\textsc{Overflow}(b)$ procedure, which re-samples a uniformly random posi-

[10] Here, $a||b$ denotes the concatenation of string a and b.

tion for the overflowed block b and inserts it back to the queue Q. (See the full version of the paper [4] for the pseudocode.)

Finally, the algorithm aborts and terminates if one of the following two events happen throughout the execution.

Abort-queue : If the size of the queue Q reaches $q_{\max}$, then the algorithm aborts and outputs ABORTQUEUE.

Abort-leaf : If the size of any leaf bucket reaches ℓ' (i.e., it becomes full), then the algorithm aborts and outputs ABORTLEAF.

This completes the description of our READ(r) algorithm; the WRITE(r, v) algorithm is defined in essentially identically the same way, except that instead of inserting b into the queue Q (in the last step of FETCH), we insert a modified b' where the content of the memory cell r (inside b) has been updated to v.

It follows by inspection that the block-path invariance is preserved by our construction. Also, note that in the above algorithm, FETCH increases the size of the queue Q by 1 and PUT-BACK is executed twice which decreases the queue size by 2. On the other hand, the FLUSH action may cause a few OVERFLOW events, and when an OVERFLOW occurs, one block will be removed from Tr and inserted to Q. Therefore, the size of the queue changes by minus one plus the number of OVERFLOW for each READ operation. The crux of our analysis is to show that the number of OVERFLOW is sufficiently small in any given (short) period of time, except with negligible probability.

We remark that throughout this algorithm's life cycle, there will be at most $\ell - 2$ non-empty blocks in each internal node except when we invoke FLUSH$(\cdot)$, in which case some intermediate states will have $\ell - 1$ blocks in a bucket (which causes an invocation of OVERFLOW).

Reducing the cache's size. We now describe how the cache can be reduced to $\operatorname{poly}\log(n)$ via recursion [27]. The key observation here is that the position map shares the same set of interfaces with our ORAM data structure. Thus, we may substitute the position map with a (smaller) ORAM of size $[n/\alpha]$. By recursively substituting the position map $O(\log n)$ times, the size of the position map will be reduced to $O(1)$.

A subtle issue here is that we need to update the position map when overflow occurs (in addition to the update for the fetched block), which results in an access to the recursive ORAM. This causes two problems. First, it reveals the time when overflow occurs, which kills obliviousness. Second, since we may make more than one recursive calls, the number of calls may blow up over $O(\log n)$ recursion levels.

To solve both problems, we instead defer the recursive calls for updating the position map to the time when we perform PUT-BACK operations. It is not hard to check that this does not hurt correctness. Recall that we do DEQUEUE once for each ORAM access, and additionally do an extra DEQUEUE for every $\log n$ ORAM accesses (to keep the cache size small). This is a deterministic pattern and hence restores obliviousness. Also note that this implies only $(\log n) + 1$ recursive calls are invoked for every $\log n$ ORAM accesses. Thus, intuitively, the blow-up rate is $(1 + (1/\log n))$ per level, and only results in a constant blow up over $O(\log n)$ levels. More precisely, consider a program execution with T ORAM access. It results in $T \cdot (1 + (1/\log n))$ access to the second ORAM, and $O(T)$ access to the final $O(1)$ size ORAM.

Now, we need to be slightly more careful to avoid the following problem. It might be possible that the one extra DEQUEUE occurs in multiple recursion levels simultaneous, resulting in unmanageable worst case running time. This problem can be avoided readily by schedule the extra DEQUEUE in different round among different recursion levels. Specifically, let $u = \log n$. For recursion level ℓ, the extra DEQUEUE is scheduled in the $(au + \ell)$-th (base-)ORAM access, for all positive integers a. Note that the extra DEQUEUE occurs in slightly slower rate in deeper recursion levels, but this will not change the asymptotic behavior of the system. As such, no two extra DEQUEUE's will be called in the same READ/WRITE operation.

On the other hand, recall that we also store the queue in the cache. We will set the queue size $q_{\max} = \operatorname{poly}\log(n)$ (specifically, we can set $q_{\max} = O(\log^{2+\varepsilon} n)$ for an arbitrarily small constant ε). Since there are only $O(\log n)$ recursion levels, the total queue size is $\operatorname{poly}\log(n)$.

3.2 Efficiency of Our ORAM

In this section, we discuss how to set the parameters of our ORAM and analyze its efficiency. We summarize the parameters of our ORAM and the setting of parameters as follows:

- ℓ: The bucket size (in terms of the number of blocks it stores) of the internal nodes of Tr. We set $\ell = \Theta(\log\log n)$.
- ℓ': The bucket size of the leaves of Tr. We set $\ell' = \Theta(\log n \log\log n)$.
- d: The depth of Tr. As mentioned, we set it to be the smallest d so that the number of leaves 2^d is at least $2(n/\alpha)/(\log n \log\log n)$.
- $q_{\max}$: The queue size. As mentioned, we set $q_{\max} = \Theta(\log^{2+\varepsilon} n)$ for an arbitrarily small constant ε.
- α: The number of memory cells in a block. As mentioned, we set α to be a constant, say 16.

We proceed to analyze the efficiency of our ORAM.

Memory overhead: constant. The external memory stores $O(\log n)$ copies of binary trees from $O(\log n)$ recursion levels. Let us first consider Tr of the top recursion level: there are $2^{d+1} - 1 = \Theta(n/\log n \log\log n)$ nodes, each of which has bucket of size at most $\ell' = \Theta(\log n \log\log n)$. The space complexity of Tr is $\Theta(n)$. As the size of Tr in each recursion level shrinks by a constant factor, one can see that the total memory overhead is constant.

Cache size: $\operatorname{poly}\log(n)$**.** As argued, the CPU cache stores the position map in the final recursion level, which has $O(1)$ size, and the queues from $O(\log n)$ recursion levels, each of which has at most $\Theta(\log^{2+\varepsilon} n)$ size. Thus, the total cache size is $O(\log^{3+\varepsilon} n)$. As we shall see below, $\operatorname{poly}\log(n)$ queue size is required in our analysis to ensure that the queue overflows with negligible probability by concentration bounds. On the other hand, we mention that our simple simulation shows that the size of the queue in the top recursion level is often well below 50 for ORAM with reasonable size.

Worst-case computational overhead: $\tilde{O}(\log^2 n)$**.** As above, we first consider the top recursion level. In the FETCH() sub-routine, we need to search from both Tr and the queue. Searching Tr requires us to traverse along a path from the root to a leaf. The time spent on each node is proportional to the size of the node's bucket. Thus, the cost here is $O(\log n \log\log n)$. One can also see searching the queue takes $O(\log n \log\log n)$ time. The total cost of $\textsc{Fetch}()$ is $O(\log n \log\log n)$.

For the $\textsc{Dequeue}()$ sub-routine, the $\textsc{Put-Back}()$ action invokes (1) one $\textsc{PopFront}()$, which takes $O(\log n \log\log n)$ time, and (2) accesses the root node, which costs $O(\log\log n)$. It also writes to the position map and triggers recursive calls. Note that certain recursive levels may execute two consecutive $\textsc{Dequeue}$'s after a $\textsc{Read}$/$\textsc{Write}$ operation. But our construction ensures only one level will execute two $\textsc{Dequeue}$'s for any $\textsc{Read}$/$\textsc{Write}$. Thus, the total cost here is $\tilde{O}(\log^2 n)$.

The $\textsc{Flush}()$ sub-routine also traverses Tr along a path, and has cost $O(\log n \log\log n)$. However, since we do a *random* number of $\textsc{Flush}()$ (according to a geometric random variable with parameter $2/3$), we only achieve *expected* $O(\log n \log\log n)$ runtime, as opposed to worst-case runtime.

To address this issue, recall that there are $O(\log n)$ recursion levels, and the total number of $\textsc{Flush}()$ is the sum of $O(\log n)$ i.i.d. random variables. Thus, the probability of performing a total of more than $\omega(\log n)$ number of $\textsc{Flush}()$'s is negligible by standard concentration result. Thus, the total time complexity is upper bounded by $\omega(\log^2 n \log\log n)$ except with negligible probability. To formally get $\tilde{O}(\log^2 n)$ worst-case computational overhead, we can add an **Abort-Flush** condition that aborts when the total number of flush in one $\textsc{Read}()$/$\textsc{Write}()$ operation exceeds some parameter $t \in \omega(\log n)$.

4 Security of Our ORAM

The following observation is central to the security of our ORAM construction (and an appropriate analogue of it was central already to the constructions of [27,5]):

> **Key observation:** *Let X denote the sum of two independent geometric random variables with mean 2. Each $Read$ and $Write$ operation traverses the tree along $X + 1$ randomly chosen paths,* independent *of the history of operations so far.*

The key observation follows from the facts that (1) just as in the schemes of [27,5], 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) we invokes the $\textsc{Flush}$ action X times and the flushing, by definition, traverses a random path, independent of the history.

Armed with the key observation, the security of our construction reduces to show that our ORAM program does not abort except with negligible probability, which follows by the following two lemmas.

Lemma 1. *Given any program Π, let $\Pi'(n, x)$ be the compiled program using our ORAM construction. We have*

$$\Pr[\textsc{AbortLeaf}] \leq \mathrm{negl}(n).$$

Proof. The proof follows by a direct application of the (multiplicative) Chernoff bound. We show 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 $\ell' + 1$ 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 $\mu = \log n \log \log n$ and by a standard multiplicative version of Chernoff bound, the probability that $\ell'+1$ elements are mapped to γ is upper bounded by $2^{-\ell'}$ when $\ell' \geq 6\mu$ (see Theorem 4.4 in [19]). By a union bound, we have that the probability of *any* node ever overflowing is bounded by $2^{-(\ell')} \cdot (n/\alpha) \cdot T$

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 $2^{-(\ell')} \cdot (n/\alpha) \cdot T \cdot \log_\alpha n$, which is negligible as long as $\ell' = c \log n \log \log n$ for a suitably large constant c. □

Lemma 2. *Given any program Π, let $\Pi'(n, x)$ be the compiled program using our ORAM construction. We have*

$$\Pr\left[\textsc{AbortQueue}\right] \leq \mathrm{negl}(n).$$

The proof of Lemma 2 is significantly more interesting. Towards proving it, in Section 5 we consider a simple variant of a "supermarket" problem (introduced by Mitzenmacher[20]) and show how to reduce Lemma 2 to an (in our eyes) basic and natural question that seems not to have been investigated before.

5 Proof of Lemma 2

We here prove Lemma 2: in Section 5.1 we consider a notion of "upset" customers in a supermarket problem [20,31,7]; in Section 5.2 we show how Lemma 2 reduced to obtaining a bound on the rate of upset customers, and in Section 5.3 we provide an upper bound on the rate of upset customers.

5.1 A Supermarket Problem

In a supermarket problem, there are D cashiers in the supermarket, all of which have empty queues in the beginning of the day. At each time step t,

- With probability $\alpha < 1/2$, an *arrival* event happens, where a new customer arrives. The new customer chooses d uniformly random cashiers and join the one with the shortest queue.
- Otherwise (*i.e.* with the remaining probability $1 - \alpha$), a *serving* event happens: a random cashier is chosen that "serves" the first customer in his queue and the queue size is reduced by one; if the queue is empty, then nothing happens.

We say that a customer is *upset* if he chooses a queue whose size exceeds some bound φ. We are interested in large deviation bounds on the number of upset customers for a given short time interval (say, of $O(D)$ or $\mathrm{poly}\log(D)$ time steps).

Supermarket problems are traditionally considered in the continuous time setting [20,31,7]. But there exists a standard connection between the continuous model and its discrete time counterpart: conditioned on the number of events is known, the continuous time model behaves in the same way as the discrete time counterpart (with parameters appropriately rescaled).

Most of the existing works [20,31,7] study only the stationary behavior of the processes, such as the expected waiting time and the maximum load among the queues over the time. Here, we are interested in large deviation bounds on a statistics over a *short* time interval; the configurations of different cashiers across the time is highly correlated.

For our purpose, we analyze only the simple special case where the number of choice $d = 1$; *i.e.* each new customer is put in a random queue.

We provide a large deviation bound for the number of upset customers in this setting.[11] .

Proposition 1. *For the (discrete-time) supermarket problem with D cashiers, one choice (i.e., $d = 1$), probability parameter $\alpha \in (0, 1/2)$, and upset threshold $\varphi \in \mathbb{N}$, for any T steps time interval $[t+1, t+T]$, let F be the number of upset customers in this time interval. We have*

$$\Pr\left[F \geq (1+\delta)(\alpha/(1-\alpha))^{\varphi} T\right] \leq \begin{cases} \exp\left\{-\Omega\left(\frac{\delta^2(\alpha/(1-\alpha))^{\varphi}T}{(1-\alpha)^2}\right)\right\} & \text{for } 0 \leq \delta \leq 1 \\ \exp\left\{-\Omega\left(\frac{\delta(\alpha/(1-\alpha))^{\varphi}T)}{(1-\alpha)^2}\right)\right\} & \text{for } \delta \geq 1 \end{cases} \tag{1}$$

Note that Proposition 1 would trivially follow from the standard Chernoff bound if T is sufficiently large (i.e., $T \gg O(D)$) to guarantee that we *individually* get concentration on each of the D queue (and then relying on the union bound). What makes Proposition 1 interesting is that it applies also in a setting when T is $\operatorname{poly}\log D$.

The proof of Proposition 1 is found in Section 5.3 and relies on a new variant Chernoff bounds for Markov chains with "resets," which may be of independent interest.

Remark 1. *One can readily translate the above result to an analogous deviation bound on the number of upset customers for (not-too-short) time intervals in the continuous time model. This follows by noting that the number of events that happen in a time interval is highly concentrated (provided that the expected number of events is not too small), and applying the above proposition after conditioning on the number of events happen in the time interval (since conditioned on the number of events, the discrete-time and continous-time processes are identical).*

5.2 From ORAM to Supermarkets

This section shows how we may apply Proposition 1 to prove Lemma 2. Central to our analysis is a simple reduction from the execution of our ORAM algorithm at level k in Tr to a supermarket process with $D = 2^{k+1}$ cashiers. More precisely, we show there exists a coupling between two processes so that each bucket corresponds with two

[11] It is not hard to see that with D cashiers, probability parameter α, and "upset" threshold φ, the expected number of upset customers is at most $(\alpha/(1-\alpha))^{\varphi} \cdot T$ for any T-step time interval.

cashiers; the load in a bucket is always upper bounded by the total number of customers in the two cashiers it corresponds to.

To begin, we need the following Lemma.

Lemma 3. *Let* $\{a_i\}_{i\geq 1}$ *be the sequence of* PUT-BACK/FLUSH *operations defined by our algorithm,* i.e. *each* $a_i \in \{\text{PUT-BACK}, \text{FLUSH}\}$ *and between any consecutive* PUT-BACK*s, the number of* FLUSH*es is a geometric r.v. with parameter* $2/3$*. Then* $\{a_i\}_{i\geq 1}$ *is a sequence of i.i.d. random variables so that* $\Pr[a_i = \text{PUT-BACK}] = \frac{1}{3}$.[12]

To prove Lemma 3, we may view the generation of $\{a_i\}_{i\geq 1}$ as generating a sequence of i.i.d. Bernoulli r.v. $\{b_i\}_{i\geq 1}$ with parameter $\frac{2}{3}$. We set a_i be a FLUSH() if and only if $b_i = 1$. One can verify that the $\{a_i\}_{i\geq 1}$ generated in this way is the same as those generated by the algorithm.

We are now ready to describe our coupling between the original process and the supermarket process. At a high-level, a block corresponds to a customer, and 2^{k+1} sub-trees in level $k + 1$ of Tr corresponds to $D = 2^{k+1}$ cashiers. More specifically, we couple the configurations at the k-th level of Tr in the ORAM program with a supermarket process as follows.

- Initially, all cashiers have zero customers.
- For each PUT-BACK(), a corresponding arrival event occurs: if a ball b with position $p = (\gamma||\eta)$ (where $\gamma \in \{0,1\}^{k+1}$) is moved to Tr, then a new customer arrives at the γ-th cashier; otherwise (*e.g.* when the queue is empty), a new customer arrives at a random cashier.
- For each FLUSH() along the path to leaf $p^* = (\gamma||\eta)$ (where $\gamma \in \{0,1\}^{k+1}$), a serving event occurs at the γ-th cashier.
- For each FETCH(), nothing happens in the experiment of the supermarket problem. (Intuitively, FETCH() translates to extra "deletion" events of customers in the supermarket problem, but we ignore it in the coupling since it only decreases the number of blocks in buckets in Tr.)

Correctness of the coupling. We shall verify the above way of placing and serving customers exactly gives us a supermarket process. First recall that both PUT-BACK and FLUSH actions are associated with uniformly random leaves. Thus, this corresponds to that at each timestep a random cashier will be chosen. Next by Lemma 3, the sequence of PUT-BACK and FLUSH actions in the execution of our ORAM algorithm is a sequence of i.i.d. variables with $\Pr[\text{PUT-BACK}] = \frac{1}{3}$. Therefore, when a queue is chosen at a new timestep, an (independent) biased coin is tossed to decide whether an arrival or a service event will occur.

Dominance. Now, we claim that at any timestep, for every $\gamma \in \{0,1\}^{k+1}$, the number of customers at γ-th cashier is at least the number of blocks stored at or above level k in Tr with position $p = (\gamma||\cdot)$. This follows by observing that (i) whenever there is a block with position $p = (\gamma||\cdot)$ moved to Tr (from PUT-BACK()), a corresponding new customer arrives at the γ-th cashier, *i.e.* when the number of blocks increase by

[12] The first operation in our system is always a PUT-BACK. To avoid that $a_1 \equiv \text{PUT-BACK}$, we can first execute a geometric number of FLUSHes before the system starts for the analysis purpose.

one, so does the number of customers, and (ii) for every FLUSH() along the path to $p^* = (\gamma||\cdot)$: if there is at least one block stored at or above level k in Tr with position $p = (\gamma||\cdot)$, then one such block will be flushed down below level k (since we flush the blocks that can be pulled down the furthest)—that is, when the number of customers decreases by one, so does the number of blocks (if possible). This in particular implies that throughout the coupled experiments, for every $\gamma \in \{0,1\}^k$ the number of blocks in the bucket at node γ is always upper bounded by the sum of the number of customers at cashier $\gamma 0$ and $\gamma 1$.

We summarize the above in the following lemma.

Lemma 4. *For every execution of our ORAM algorithm (i.e., any sequence of* READ *and* WRITE *operations), there is a coupled experiment of the supermarket problem such that throughout the coupled experiments, for every $\gamma \in \{0,1\}^k$ the number of blocks in the bucket at node γ is always upper bounded by the sum of the number of customers at cashier $\gamma 0$ and $\gamma 1$.*

From Lemma 4 and Proposition 1 to Lemma 2. Note that at any time step t, if the queue size is $\leq \frac{1}{2}\log^{2+\epsilon} n$, then by Proposition 1 with $\varphi = \ell/2 = O(\log\log n)$ and Lemma 4, except with negligible probability, at time step $t + \log^4 n$, there have been at most $\omega(\log n)$ overflows per level in the tree and thus at most $\frac{1}{2}\log^{2+\epsilon} n$ in total. Yet during this time "epoch", $\log^3 n$ element have been "popped" from the queue, so, except with negligible probability, the queue size cannot exceed $\frac{1}{2}\log^{2+\epsilon} n$.

It follows by a union bound over $\log^3 n$ length time "epochs", that except with negligible probability, the queue size never exceeds $\log^{2+\epsilon} n$.

5.3 Analysis of the Supermarket Problem

We now prove Proposition 1. We start with interpreting the dynamics in our process as evolutions of a Markov chain.

A Markov Chain Interpretation. In our problem, at each time step t, a random cashier is chosen and either an arrival or a serving event happens at that cashier (with probability α and $(1-\alpha)$, respectively), which increases or decreases the queue size by one. Thus, the behavior of each queue is governed by a simple Markov chain M with state space being the size of the queue (which can also be viewed as a drifted random walk on a one dimensional finite-length lattice). More precisely, each state $i > 0$ of M transits to state $i+1$ and $i-1$ with probability α and $(1-\alpha)$, respectively, and for state 0, it transits to state 1 and stays at state 0 with probability α and $(1-\alpha)$, respectively. In other words, the supermarket process can be rephrased as having D copies of Markov chains M, each of which starts from state 0, and at each time step, one random chain is selected and takes a move.

We shall use Chernoff bounds for Markov chains [9,14,16,3] to derive a large deviation bound on the number of upset customers. Roughly speaking, Chernoff bounds for Markov chains assert that for a (sufficiently long) T-steps random walk on an ergodic finite state Markov chain M, the number of times that the walk visits a subset V of states is highly concentrated at its expected value $\pi(V) \cdot T$, provided that the chain M has spectral expansion[13] $\lambda(M)$ bounded away from 1. However, there are a few technical issues, which we address in turn below.

Overcounting. The first issue is that counting the number of visits to a state set $V \subset S$ does not capture the number of upset customers exactly—the number of upset customers corresponds to the *number of transits* from state i to $i+1$ with $i+1 \geq \varphi$. Unfortunately, we are not aware of Chernoff bounds for counting the number of transits (or visits to an edge set). Nevertheless, for our purpose, we can set $V_\varphi = \{i : i \geq \varphi\}$ and the number of visits to V_φ provides an *upper bound* on the number of upset customers.

Truncating the chain. The second (standard) issue is that the chain M for each queue of a cashier has infinite state space $\{0\} \cup \mathbb{N}$, whereas Chernoff bounds for Markov chains are only proven for finite-state Markov chains. However, since we are only interested in the supermarket process with finite time steps, we can simply truncate the chain M at a sufficiently large K (say, $K \gg t+T$) to obtain a chain M_K with finite states $S_K = \{0, 1, \ldots, K\}$; that is, M_K is identical to M, except that for state K, it stays at K with probability α and transits to $K-1$ with probability $1-\alpha$. Clearly, as we only consider $t+T$ time steps, the truncated chain M_K behaves identical to M. It's also not hard to show that M_K has stationary distribution π_K with $\pi_K(i) = (1-\beta)\beta^i/(1-\beta^{K+1})$, and spectral gap $1-\lambda(M_K) \geq \Omega(1/(1-\alpha)^2)$.[14]

Correlation over a short time frame. The main challenge, however, is to establish large deviation bounds for a *short* time interval T (compared to the number D of chains). For example, $T = O(D)$ or even $\operatorname{poly}\log(D)$, and in these cases the expected number of steps each of the D chains take can be a small constant or even $o(1)$. Therefore, we cannot hope to obtain meaningful concentration bounds individually for each single chain. Finally, the D chains are not completely independent: only one chain is selected at each time step. This further introduces correlation among the chains.

We address this issue by relying on a new variant of Chernoff bounds for Markov chains with "resets," which allows us to "glue" walks on D separate chains together and yields a concentration bound that is as good as a T-step random walk on a single chain. We proceed in the following steps.

– Recall that we have D copies of truncated chains M_K starting from state 0. At each time step, a random chain is selected and we takes one step in this chain. We want to upper bound the total number of visits to V_φ during time steps $[t+1, t+T]$.
– We first note that, as we are interested in upper bounds, we can assume that the chains start at the stationary distribution π_K instead of the 0 state (i.e., all queues have initial size drawn from π_K instead of being empty). This follows by noting that starting from π_K can only increase the queue size *throughout* the whole process for *all* of D queues, compared to starting from empty queues, and thus the number of visits to V_φ can only increase when starting from π_K in compared to starting from state 0 (this can be formalized using a standard coupling argument).
– Since walks from the stationary distribution remain at the stationary distribution, we can assume w.l.o.g. that the time interval is $[1, T]$. Now, as a thought experiment, we

[13] For an ergodic reversible Markov chain M, the *spectral expansion* $\lambda(M)$ of M is simply the second largest eigenvalue (in absolute value) of the transition matrix of M. The quantity $1-\lambda(M)$ is often referred to as the spectral gap of M.

[14] One can see this by lower bounding the conductance of M_K and applying Cheeger's inequality.

can decompose the process as follows. We first determine the number of steps each of the D chains take during time interval $[1, T]$; let c_j denote the number of steps taken in the j-th chain. Then we take c_j steps of random walk from the stationary distribution π_K for each copy of the chain M_K, and count the total number of visit to V_φ.

– Finally, we can view the process as taking a T-step random walk on M_K with "resets." Namely, we start from the stationary distribution π_K, take c_1 steps in M_K, "reset" the distribution to stationary distribution (by drawing an independent sample from π_K) and take c_2 more steps, and so on. At the end, we count the number of visits to V_φ, denoted by X, as an upper bound on the number of upset customers.

Intuitively, taking a random walk with resets injects additional randomness to the walk and thus we should expect at least as good concentration results. We formalize this intuition as the following Chernoff bound for Markov chains with "resets"—the proof of which follows relatively easy from recent Chernoff bounds for Markov chains [3] and is found Section 5.4—and use it to finish the proof of Proposition 1.

Theorem 2 (Chernoff Bounds for Markov Chains with Resets)**.** *Let M be an ergodic finite Markov chain with state space S, stationary distribution π, and spectral expansion λ. Let $V \subset S$ and $\mu = \pi(V)$. Let $T, D \in \mathbb{N}$ and $1 = T_0 \leq T_1 \leq \cdots \leq T_D < T_{D+1} = T + 1$. Let $(W_1, \ldots, W_T)$ denote a T-step random walk on M from stationary with resets at steps $T_1, \ldots, T_D$; that is, for every $j \in \{0, \ldots, D\}$, $W_{T_j} \leftarrow \pi$ and $W_{T_j+1}, \ldots, W_{T_{j+1}-1}$ are random walks from W_{T_j}. Let $X_i = 1$ iff $W_i \in V$ for every $i \in [T]$ and $X = \sum_{i=1}^{T} X_i$. We have*

$$\Pr\left[X \geq (1+\delta)\mu T\right] \leq \begin{cases} \exp\left\{-\Omega(\delta^2(1-\lambda)\mu T)\right\} & \text{for } 0 \leq \delta \leq 1 \\ \exp\left\{-\Omega(\delta(1-\lambda)\mu T)\right\} & \text{for } \delta \geq 1 \end{cases}$$

Now, recall that $1 - \lambda(M_K) = \Omega(1/(1-\alpha)^2)$ and $\pi_K(\varphi) = \beta^\varphi/(1-\beta^{K+1}) = (\alpha/1-\alpha)^\varphi/(1-\beta^{K+1})$. Theorem 2 says that for every possible $c_1, \ldots, c_D$ (corresponding to resetting time $T_j = \sum_{l=1}^{j} c_j + 1$),

$$\Pr\left[X \geq \frac{(1+\delta)(\alpha/1-\alpha)^\varphi T}{(1-\beta^{K+1})} \middle| c_1, \ldots, c_D\right] \leq \begin{cases} \exp\left\{-\Omega\left(\frac{\delta^2(\alpha/1-\alpha)^\varphi T}{(1-\alpha)^2(1-\beta^{K+1})}\right)\right\} & \text{for } 0 \leq \delta \leq 1 \\ \exp\left\{-\Omega\left(\frac{\delta(\alpha/1-\alpha)^\varphi T)}{(1-\alpha)^2(1-\beta^{K+1})}\right)\right\} & \text{for } \delta \geq 1 \end{cases}$$

Since X is an upper bound on the number of upset customers, and the above bound holds for every $c_1, \ldots, c_D$ and for every $K \geq t + T$, Proposition 1 follows by taking $K \to \infty$. □

5.4 Chernoff Bounds for Markov Chains with Reset

We now prove Theorem 2. The high level idea is simple—we simulate the resets by taking a sufficiently long "dummy" walk, where we "turn off" the counter on the number of visits to the state set V. However, formalizing this idea requires a more general version of Chernoff bounds that handles "time-dependent weight functions," which allows us to turn on/off the counter. Additionally, as we need to add long dummy walks, a multiplicative version (as opposed to an additive version) Chernoff bound is needed to derive meaningful bounds. We here rely on a recent generalized version of Chernoff bounds for Markov chains due to Chung, Lam, Liu and Mitzenmacher [3].

Theorem 3 ([3]). *Let M be an ergodic finite Markov chain with state space S, stationary distribution π, and spectral expansion λ. Let $\mathcal{W} = (W_1, \ldots, W_T)$ denote a T-step random walk on M starting from stationary distribution π, that is, $W_1 \leftarrow \pi$. For every $i \in [T]$, let $f_i : S \to [0,1]$ be a weight function at step i with expected weight $\mathbb{E}_{v \leftarrow \pi}[f_i(v)] = \mu_i$. Let $\mu = \sum_i \mu_i$. Define the total weight of the walk $(W_1, \ldots, W_t)$ by $X \triangleq \sum_{i=1}^{t} f_i(W_i)$. Then*

$$\Pr\left[X \geq (1+\delta)\mu\right] \leq \begin{cases} \exp\left\{-\Omega(\delta^2(1-\lambda)\mu)\right\} & \text{for } 0 \leq \delta \leq 1 \\ \exp\left\{-\Omega(\delta(1-\lambda)\mu)\right\} & \text{for } \delta > 1 \end{cases}$$

We now proceed to prove Theorem 2.

Proof of Theorem 2. We use Theorem 3 to prove the theorem. Let $f : S \to [0,1]$ be an indicator function on $V \subset S$ (i.e., $f(s) = 1$ iff $s \in V$) .The key component from Theorem 3 we need to leverage here is that the functions f_i can change over the time. Here, we shall design a very long walk $\mathcal{V}$ on M so that the marginal distribution of a specific collections of "subwalks" from $\mathcal{V}$ will be statistically close to $\mathcal{W}$. Furthermore, we design $\{f_i\}_{i \geq 0}$ in such a way that those "unused" subwalks will have little impact to the statistics we are interested in. In this way, we can translate a deviation bound on $\mathcal{V}$ to a deviation bound on $\mathcal{W}$. Specifically, let $T(\epsilon)$ be the mixing time for M (*i.e.* the number of steps needed for a walk to be ϵ-close to the stationary distribution in statistical distance). Here, we let $\epsilon \triangleq \exp(-DT)$ (ϵ is chosen in an arbitrary manner so long as it is sufficiently small). Given $1 = T_0 \leq T_1 \leq \cdots \leq T_D < T_{D+1} = T+1$, we define $\mathcal{V}$ and f_i as follows: $\mathcal{V}$ will start from π and take $T_1 - 2$ steps of walk. In the mean time, we shall set $f_i = f$ for all $i < T_1$. Then we "turn off" the function f_i while letting $\mathcal{V}$ keep walking for $T(\epsilon)$ more steps, *i.e.* we let $f_i = 0$ for all $T_1 \leq i \leq T_1 + T(\epsilon) - 1$. Intuitively, this means we let $\mathcal{V}$ take a long walk until it becomes close to π again. During this time, f_i is turned off so that we do not keep track of any statistics. After that, we "turn on" the function f_i again for the next $T_2 - T_1$ steps (*i.e.* $f_i = f$ for all $T_1 + T(\epsilon) \leq i \leq T_2 + T(\epsilon) - 1$, followed by turning f_i off for another $T(\epsilon)$ steps. We continue this "on-and-off" process until we walk through all T_j's.

Let $\mathcal{V}'$ be the subwalks of $\mathcal{V}$ with non-zero f_i. One can see that the statistical distance between $\mathcal{V}'$ and $\mathcal{W}$ is $\mathrm{poly}(D,T)\exp(-DT) \leq \exp(-T + o(T))$. Thus, for any θ we have

$$\begin{aligned} \Pr\left[\textstyle\sum_{w \in \mathcal{W}} f(w) \geq \theta\right] &\leq \Pr\left[\textstyle\sum_{v' \in \mathcal{V}'} f(v') \geq \theta\right] + \exp(-T + o(T)) \\ &= \Pr\left[\textstyle\sum_{v \in \mathcal{V}} f(v) \geq \theta\right] + \exp(-T + o(T)). \end{aligned} \tag{2}$$

By letting $\theta = (1+\delta)\mu T$ and using Theorem 3 to the right hand side of (2), we finish our proof. $\square$

6 Acknowledgements

We are extremely grateful to an anonymous reviewer for pointing out a subtle missing implementation detail needed to make the recursion go through.

References

1. Miklós Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In *STOC*, pages 181–190, 2010. 1, 2
2. Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. CSAIL Technical Report: MIT-CSAIL-TR-2011-018, 2012. 1
3. K. M. Chung, H. Lam, Z. Liu, and M. Mitzenmacher. Chernoff-Hoeffding bounds for Markov chains: Generalized and simplified. In *Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science (STACS)*, 2012. 3, 14, 16, 17
4. Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. *CoRR*, abs/1307.3699, 2013. 8
5. Kai-Min Chung and Rafael Pass. A simple ORAM. Cryptology ePrint Archive, Report 2013/243, 2013. 2, 3, 4, 6, 10
6. Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011. 1, 2
7. Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Eng.*, 12(5):662–675, 1986. 11, 12
8. Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies*, pages 1–18, 2013. 2, 6
9. D. Gillman. A Chernoff bound for random walks on expander graphs. *SIAM Journal on Computing*, 27(4), 1997. 3, 14
10. Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987. 1, 4
11. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996. 1, 4, 5
12. Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP (2)*, pages 576–587, 2011. 1, 2
13. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, pages 157–167, 2012. 1, 2
14. N. Kahale. Large deviation bounds for Markov chains. *Combinatorics, Probability, and Computing*, 6(4), 1997. 3, 14
15. 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, 2
16. P. Lezaud. Chernoff-type bound for finite Markov chains. *Annals of Applied Probability*, 8(3):849–867, 1998. 3, 14
17. Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396, 2013. 2
18. Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. CCS '13, pages 311–324. ACM, 2013. 4
19. M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. 11

20. Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001. 3, 11, 12
21. Michael Mitzenmacher, Balaji Prabhakar, and Devavrat Shah. Load balancing with memory. In *FOCS*, pages 799–808, 2002. 3
22. Michael Mitzenmacher and Salil Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 746–755, 2008. 20
23. Michael Mitzenmacher and Berhold Vocking. The asymptotics of selecting the shortest of two, improved. In *Proceedings of the Annual Allerton Conference on Communication Control and Computing*, volume 37, pages 326–327, 1999. 3
24. Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997. 1
25. Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519, 2010. 1, 2
26. Devavrat Shah and Balaji Prabhakar. The use of memory in randomized load balancing. In *Information Theory, 2002. Proceedings. 2002 IEEE International Symposium on*, page 125. IEEE, 2002. 3
27. 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, 4, 5, 6, 8, 10
28. Emil Stefanov and Elaine Shi. Path O-RAM: An extremely simple oblivious RAM protocol. *CoRR*, abs/1202.5150v1, 2012. 2, 3, 4
29. Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *NDSS*, 2012. 1
30. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path O-RAM: An extremely simple oblivious RAM protocol. In *CCS*, 2013. 4
31. Nikita Dmitrievna Vvedenskaya, Roland L'vovich Dobrushin, and Fridrikh Izrailevich Karpelevich. Queueing system with selection of the shortest of two queues: An asymptotic approach. *Problemy Peredachi Informatsii*, 32(1):20–34, 1996. 3, 11, 12
32. Peter Williams and Radu Sion. Usable PIR. In *NDSS*, 2008. 1
33. Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*, pages 139–148, 2008. 1

A Implementation details.

This section discusses a number of implementation details in our algorithm.

The queue at the cache. We now describe how we may use a hash table and a standard queue (that could be encapsulated in commodity chips) to implement our queue with slightly non-standard behavior, which still suffices for our ORAM. Here, we only assume the hash table uses universal hash function and it resolves collisions by using a linked-list. To implement the $\textsc{Insert}(\text{Block}\ :b)$ procedure, we simply insert b to both the hash table and the queue. The key we use is b's value at the position map. Doing so we may make sure the maximum load of the hash table is $O(\log n)$ whp [22]. To implement $\textsc{Find}(\text{int}\ :i, \text{word}\ :p)$, we find the block b from the hash table. If it exists, return the block and delete it. However, for simplicity of implementation, we *do not* delete b at the queue. This introduces inconsistencies between the hash table and the queue, which we take care below in $\textsc{PopFront}()$.

We now describe how we implement $\textsc{PopFront}()$. Here, we need to be careful with the inconsistencies. We first pop a block from the queue. Then we need to check whether the block is in hash table. If not, that means the block was already deleted earlier. In this case, $\textsc{PopFront}()$ will not return anything (because we need a hard bound on the running time). Note that this does not effect the correctness of our analysis, since the queue size is indeed decreased by 1 for every $\textsc{Put-Back}()$ action.

One can see that the above implementation relies only on standard hash table and queue, and $\textsc{Insert}()$ takes $O(1)$ time and the other two operations take $\omega(\log n)$ time (except with negligible probability).

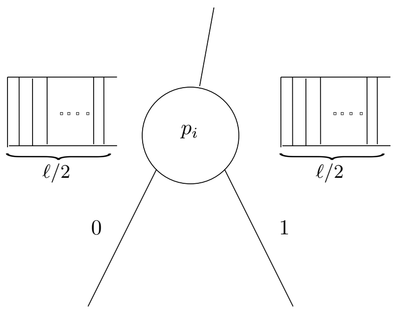

Fig. 1. In the $\textsc{Flush}$ operation, we may imagine each bucket is splitted into two sub-arrays so that blocks that will travel to different subtrees are stored in different arrays. An overflow occurs when either sub-array's size reaches $\frac{\ell}{2}$.