Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound

Xiao Shaun Wang
wangxiao@cs.umd.edu
University of Maryland

T-H. Hubert Chan
hubert@cs.hku.hk
University of Hong Kong

Elaine Shi
elaine@cs.umd.edu
University of Maryland

Abstract

Oblivious RAM (ORAM) constructions have traditionally been measured by their bandwidth cost, or the blowup in the ORAM's running time in comparison with the non-oblivious baseline. While these metrics can suitably characterize an ORAM's performance in secure processor and cloud outsourcing applications, recent works have observed that other applications such as secure multi-party computation demand a different metric, namely, the ORAM's circuit complexity.

Following the tree-based ORAM paradigm by Shi et al., we propose a new ORAM scheme called Circuit ORAM. Circuit ORAM achieves $O(D \log N)\omega(1)$ total circuit size[1] (over all protocol interactions) for memory words of $D = \Omega(\log^2 N)$ bits, while achieving a negligible failure probability. For memory words of $D = \Omega(\log^2 N)$ bits, Circuit ORAM achieves smaller circuits both asymptotically and in practice than all previously known ORAM schemes. Empirical results suggest that Circuit ORAM yields circuits that are 8x to 48x smaller than Path ORAM for datasets of roughly 1GB. The speedup will be even greater for larger data sizes.

Circuit ORAM is also theoretically interesting when interpreted under the traditional metrics. Parameterizing the scheme slightly differently, we show the following. Let $0 < \epsilon < 1$ denote any constant, and consider a family of RAMs with N words each of which N^ϵ bits in size. Any RAM in this class can be compiled to an Oblivious RAM with $O(1)$ words of CPU cache, running in $O(T \log N)\omega(1)$ time, and achieving negligible statistical failure probability (or running in $O(T \log N)$ time but with inverse polynomial failure probability). This suggests that certain stronger interpretations of the Goldreich-Ostrovsky ORAM lower bound are tight — in particular their lower bound trivially generalizes to any $O(1)$ failure probability, and works for arbitrary memory word sizes.

1 Introduction

Oblivious RAM (ORAM), initially proposed by Goldreich and Ostrovsky [16,18], is a general cryptographic primitive that allows oblivious accesses to sensitive data, such that access patterns during the computation reveal no secret information. Since the original proposal of ORAM [18], it has been studied in various application settings including secure processors [8–10,35,41], cloud outsourced storage [19,43,44,51] and secure multi-party computation [11,12,23,26,30,47].

1.1 ORAM with Small Circuit Complexity

Traditionally, an ORAM scheme is measured by its bandwidth cost [19,42,44–46] or the blowup in the RAM's running time [16,18]. While these metrics are suitable for characterizing an ORAM's performance in the secure processor and cloud outsourced storage settings, recently, Wang *et al.* [47] observe that in cryptographic applications such as secure multi-party computation [17,53] or (reusable) Garbled RAM [13–

[1]We use the notation $g(N) = O(f(N))\omega(1)$ to denote that for any $\alpha(N) = \omega(1)$, it holds that $g(N) = O(f(N)\alpha(N))$.

Scheme	Circuit Size	Security
Hierarchical ORAMs		
Goldreich-Ostrovsky [16,18]	$O(D \log^3 N + C_{\mathrm{PRF}} \log^2 N)$	Comp.
Goodrich-Mitzenmacher [19]	$O(D \log^2 N + C_{\mathrm{PRF}} \log N)$	Comp.
Kushilevitz, Lu, and Ostrovsky [29]	$O((D + C_{\mathrm{PRF}}) \cdot \log^2 N / \log \log N)$	Comp.
Lu and Ostrovsky [32] (Note: 2-server model)	$O((D + C_{\mathrm{PRF}}) \cdot \log N)$	Comp.
Tree-based ORAMs		
Binary-tree ORAM [42]	$O((D + \log^2 N) \log^2 N)\omega(1)$	Stat.
Chung, Liu, and Pass [5] (naive circuit)	$O((D + \log^2 N) \log^3 N)\omega(1)$	Stat.
Chung, Liu, and Pass [5] (w/ oblivious queue [36,40,54])	$O((D + \log^2 N) \log^2 N)\omega(1)$	Stat.
Path ORAM (naive circuit) [46]	$O((D + \log^2 N) \log^2 N)\omega(1)$	Stat.
Path ORAM (o-sort circuit) [47]	$O((D + \log^2 N) \log N \log \log N)\omega(1)$	Stat.
Circuit ORAM (This Paper)	$\mathbf{O((D + \log^2 N) \log N)\boldsymbol{\omega}(1)}$	**Stat.**

Table 1: **Circuit size of various ORAM schemes.** All schemes are parameterized to have $\frac{1}{N^{\omega(1)}}$ failure probability. The variable C_{PRF} denotes the circuit size of a PRF function with input size of $O(\log N)$ bits. Among all single-server ORAM schemes, Circuit ORAM has asymptotically the smallest circuit size if C_{PRF} is at least $\omega(\log N \log \log N)$ — which is true for all known PRF constructions provably secure based on computationally hard problems. When the memory word size is $\Omega(\log^2 N)$, Circuit ORAM's circuit size is competitive with Lu and Ostrovsky's multi-server ORAM [32] asymptotically, and in practice orders of magnitude better.

15, 33, 34], the *circuit complexity* of the ORAM scheme is a more appropriate metric. Specifically, Wang *et al.* define the circuit complexity of an ORAM scheme to be the total size of ORAM circuits over all protocol interactions (when we express the ORAM algorithm in smallest possible circuits). In this paper, all circuits are assumed to be boolean circuits with constant fan-in and fan-out, with AND and XOR gates. Besides introducing the circuit complexity metric, Wang *et al.* also point out the following:

- Among known constructions, Path ORAM has the smallest circuit size for reasonably large memory word sizes. Specifically, they demonstrate this by implementing Path ORAM with circuits of $O(D \log N \log \log N) \cdot \omega(1)$ total size for memory words of $D = \Omega(\log^2 N)$ bits.
- Additionally, Wang *et al.* give a heuristic tree-based construction called SCORAM [47], and show that empirically SCORAM achieves roughly 10x improvement in circuit size in comparison with the state-of-the-art Path ORAM [46]. Unfortunately, SCORAM did not have a formal security proof; further, empirical simulations of SCORAM suggest that SCORAM's circuit complexity is asymptotically worse than that of Path ORAM (while in practice SCORAM is 10x better for moderately large data sizes).

The work by Wang *et al.* leaves open the intriguing question of how to design ORAM schemes with asymptotically smaller circuit complexity. Following the tree-based ORAM paradigm by Shi *et al.* [42], we construct a novel ORAM scheme called Circuit ORAM. For a memory word size of $D = \Omega(\log^2 N)$, Circuit ORAM achieves $O(D \log N) \cdot \omega(1)$ circuit complexity. Notably, Circuit ORAM outperforms all existing (single-server) ORAM schemes in terms of circuit complexity both *asymptotically* and *in practice.* A more detailed asymptotical comparison is provided in Table 1 (note that existing hierarchical ORAMs rely on PRFs. All existing provably secure PRFs require $\omega(\log N \log \log N)$ cost for input size of $O(\log N)$ bits). On the practical front, our empirical results suggest that Circuit ORAM outperfoms Path ORAM by 8x to 48x in terms of circuit size at a 1GB dataset size, and outperforms SCORAM by 4x — this speedup will be even greater for bigger datasets. Like most other tree-based ORAMs, Circuit ORAM is *statistically secure*, i.e., security holds against a computationally unbounded adversary.

Remark 1 (Model) *For showing that certain stronger interpretations of the Goldreich-Ostrovsky lower* *bound are tight, we will use the standard ORAM model (with a uniform memory word size). However, the remainder of paper will mostly consider a slight variant of the standard ORAM model. For a given RAM with N memory words, each D bits in length, we assume that its ORAM counterpart is a RAM with non-uniform memory word sizes. This means that we allow the ORAM to have memory words of different sizes.*

1.2 When the Goldreich-Ostrovsky Lower Bound is Tight

Goldreich and Ostrovsky [16, 18] show that for any N-word RAM, its oblivious simulation must incur at least $\Omega(\log N)$ blowup in the RAM's running time — assuming $O(1)$ words of CPU cache. That is, if the non-oblivious RAM ran in time T, then its oblivious counterpart would run in time $\Omega(T \log N)$. Whether this lower bound is tight has been one of the biggest open questions in this line of research for the past twenty-seven years [16].

Circuit ORAM can also be recast in terms of traditional ORAM metrics. As an interesting by-product of our work, we show that certain stronger interpretations of the famous Goldreich-Ostrovsky lower bound are tight. This gives a *partial* solution to this open question in the following sense.

First, observe that while not made explicit in their theorem statement [16,18], Goldreich and Ostrovsky's lower bound proof clearly shows that the $\Omega(\log N)$ lower bound holds even for any constant statistical failure probability, and for arbitrary memory word sizes, as formally stated below:

Theorem 1 (Goldreich-Ostrovsky ORAM lower bound [16, 18], a stronger interpretation) *For any N-word RAM with arbitrary memory word size and running in time T, its oblivious simulation must incur at least $\Omega(T \log N)$ run-time, even when tolerating up to any $O(1)$ failure probability against a computationally unbounded adversary.*

In fact, the Goldreich-Ostrovsky lower bound can be interpreted in even stronger ways. Notably, their lower bound proof does not account for the cost of the CPU-caching and transmission of metadata (i.e., any helper data structures that are independent of the actual contents of memory words). Therefore, their lower bound holds even when all metadata caching and transmission comes for free! In this sense, the Goldreich-Ostrovsky lower bound is very powerful. Ideally, one wishes to show that even weaker interpretations of the lower bound are tight. In this paper, we are only able to show the tightness of certain stronger interpretations.

Circuit ORAM can easily be parameterized and viewed in a different light, as stated in the following informal theorem:

Theorem 2 (Informal.) *Let $0 < \epsilon < 1$ denote an arbitrary constant. For any N-word RAM running in time T where each word is $D = O(N^\epsilon)$ bits in size, there exists an ORAM counterpart running in time $O(T \log N)$, consuming $O(1)$ words of CPU cache, and with a* **uniform** *memory word size of $D \cdot (1 + o(1))$. Further, the ORAM counterpart tolerates inverse polynomial statistical failure probability.*

Hence, the stronger interpretation of the Goldreich-Ostrovsky lower bound as stated in Theorem 1 is tight.

As will soon become clear, the large memory word size assumption is to allow $O(1)$ depths of recursion using the standard recursion trick of tree-based ORAMs [42].

It is also helpful to compare Circuit ORAM with the state-of-the-art Path ORAM [46] in terms of the traditional metric, i.e., blowup in the ORAM's running time. In comparison, Path ORAM requires $\Omega(\log N)$ words of CPU cache to achieve the same asymptotic overhead in terms of blowup of the ORAM's runtime. This is necessary since in Path ORAM, the CPU must locally store the the eviction path and the stash to perform the eviction algorithm in $O(\log N)$ time. In contrast, Circuit ORAM requires only $O(1)$ blocks of CPU cache.

Regarding the tightness of the Goldreich-Ostrovsky lower bound, even though the partial answers we give are not yet entirely satisfying, we are nevertheless the first ones to show that any interpretation of the Goldreich-Ostrovsky lower bound (or its proof) is tight. Our work leaves open the intriguing question whether weaker interpretations of the Goldreich-Ostrovsky lower bound are tight. A tighter lower bound, however, can only be shown if 1) we consider a model that tolerates negligible or no statistical failures; or 2) the lower bound proof explicitly makes use of small block size assumptions.

Scheme	**Amortized Bandwidth Cost**	**Client Storage**	**Server Storage**
	Hierarchical ORAMs		
Goldreich-Ostrovsky [18]	$O(D \log^3 N)$	$O(D)$	$O(DN \log N)$
Goodrich-Mitzenmacher [19]	$O(D \log^2 N)$	$O(D)$	$O(DN)$
Kushilevitz *et al.* [29]	$O(D \log^2 N / \log\log N)$	$O(D)$	$O(DN)$
Lu and Ostrovsky [32] (Note: 2-server model)	$O(D \log N)$	$O(D)$	$O(DN)$
	Tree-based ORAMs		
Binary-tree ORAM [42]	$O(D \log^2 N + \log^4 N) \cdot \omega(1)$	$O(D)$	$O(DN \log N) \cdot \omega(1)$
Chung *et al.* [5]	$O\left(\left(D \log N + \log^3 N\right) \log\log N\right)$	$O(D \log^2 N) \cdot \omega(1)$	$O(DN \log N \log\log N)$
Path ORAM(naive circuit) [46]	$O(D \log N + \log^3 N)$	$O(D \log N) \cdot \omega(1)$	$O(DN)$
Path ORAM(o-sort circuit) [47]	$O\left(\left(D \log N + \log^3 N\right) \log\log N\right) \cdot \omega(1)$	$O(D)$	$O(DN)$
Circuit ORAM (This paper)	$O(D \log N + \log^3 N) \cdot \omega(1)$	$O(D)$	$O(DN)$

Table 2: **Comparison of various ORAMs in terms of bandwidth cost.** The bounds are expressed for negligible failure probabilities. For all ORAMs based on the tree-based framework, their bandwidth cost includes two parts, a part for transferring blocks, and a part for transferring metadata. The metadata part would be absorbed into the other term if $D = \Omega(\log^2 N)$.

1.3 Technical Highlights

Terminology. For convenience, in this paper, we use the terms "memory word", "word", or "block" interchangably. Further, we use the terms "CPU" and "ORAM client" interchangably. We also use the terms "memory" and "ORAM server" interchangably.

Path ORAM has a complex eviction circuit. Wang *et al.* show that Path ORAM can be implemented with circuits of $O(D \log N \log N \log N) \cdot \omega(1)$ total size (for negligible security failures). Our goal is to remove the $\log\log N$ factor. One immediate question is whether we can implement Path ORAM itself with a smaller circuit. This seems inherently difficult, since it is not hard to see that Path ORAM's eviction algorithm implies sorting for a path of $\Theta(\log N)$ length (imagine that bucket size were 1). Therefore, the $\log\log N$ gap for Path ORAM seems inherent.

Reducing eviction complexity. Our idea is to find an eviction circuit that is less complex than that of Path ORAM's, and yet preserving the effectiveness of eviction. Achieving this is non-trivial. We first tested numerous ideas empirically, most of which failed to empirically bound the stash size since the eviction algorithm is not as aggressive as Path ORAM. After months of trying, we eventually identified a good empirical candidate, which in turn inspired the design of its provable variant, Circuit ORAM, that is documented in this paper. Just like Path ORAM [46] and its variants [9, 12], Circuit ORAM performs eviction on $O(1)$ number of paths upon each data access. Our key idea is to complete the eviction algorithm within a single block scan of the current eviction path (while evicting as aggressively as we can). Achieving this directly introduces some difficulties due to a "lack-of-foresight" problem, i.e., the ORAM client does not know when to pick up a block and remove it from the path, and when to drop it into an empty slot on the path. To tackle this problem, we leveraged two additional metadata scans to precompute the foresight required, before beginning the real block scan. Our construction and proofs borrow ideas from Path ORAM [46] and the CLP ORAM [5]. And yet our construction and proofs differ in a substantial and non-trivial manner from either.

1.4 Related Work

Hierarchical ORAMs. As mentioned earlier, Oblivious RAM was first proposed in a groundbreaking work by Goldreich and Ostrovsky [18]. In addition to the aforementioned lower bound, Goldreich and Ostrovsky were the first to propose a poly-logarithmic hierarchical construction, which was subsequently improved in numerous works [4, 16, 18–22, 29, 31, 32, 37–39, 48–51]. In particular, Goodrich and Mitzenmacher [19] achieves bandwidth cost of $O(D \log^2 N)$ under constant client storage and cost of $O(D \log N)$ under $O(N^\epsilon)$ client storage for $0 < \epsilon < 1$. Later, Lu *et al.* show that $O(D \log^2 N / \log\log N)$ bandwidth cost can be achieved under $O(D)$ client storage [29]. Further, hierarchical ORAMs with $O(\sqrt{N})$ client-side storage have been applied to secure outsourced storage applications [19, 22, 48, 51, 52].

Tree-based ORAM framework. The tree-based ORAM framework, initially proposed by Shi *et al.* [42], departs fundamentally from the hierarchical framework [18], and is a new paradigm for constructing a class of ORAM schemes. Several later works [5, 11, 46] improved Shi *et al.*'s initial construction [42] (commonly referred to as binary-tree ORAM). These schemes are conceptually simpler, statistically secure, and easy to implement in secure processors [8–10, 35, 41] or secure computation [12, 23, 26, 30, 47]. A more detailed description of tree-based ORAMs are provided in Section 2.

Remarks about the ORAM lower bound. Besides efforts at constructing more efficient upper bounds, the community has also been interested in tightening the lower bound. Beame and Machmouchi [3] show a super-logarithmic lower bound for oblivious branching programs. Although some works cited their lower-bound as being applicable to the ORAM setting [6, 19], it was later recognized that **Beame *et al.*'s super-logarithmic lower bound is not applicable to the standard model of Oblivious ORAM.** As the authors noted themselves in an updated version [3], one key difference is that the standard ORAM model requires that the probability distribution of the observed access patterns be statistically close regardless of the input; whereas Beame's model requires that for each given random string r, the access pattern be independent of the input. Therefore, their super-logarithmic lower-bound is in a much stronger model than standard ORAM, and hence inapplicable to ORAM. To date, Goldreich and Ostrovsky's original lower bound is still the best we know; and this is for a good reason: their lower bound is indeed tight in their model. One subtlety regarding the Goldreich-Ostrovsky lower bound was pointed out by Apon *et al.* [2], who show that the lower bound holds only when the storage server (i.e., memory) is passive and incapable of computation, and is no longer applicable when the storage server can actively perform computation. Apon *et al.* [2] referred to the latter model with active server computation as Verifiable Oblivious Storage.

Additional related work. Williams and Sion propose schemes for cloud storage outsourcing while preserving access pattern privacy [48, 51]. Their scheme works in a slightly different "Verifiable Oblivious Storage" model as defined by Apon *et al.* [2], since it relies on the server to perform active computation to support hash table queries. This model is incomparable and not subject to Goldreich-Ostrovsky's lower bound. Later, Williams *et al.* also relied on these ideas to further design a single-roundtrip verifiable oblivious storage system [49], and implemented an oblivious file systems [52].

Lu and Ostrovsky propose a two-server ORAM which has $O(D \log N)$ bandwidth cost under $O(D)$ client storage. Stefanov *et al.* propose a partition-based ORAM framework that partitions a big ORAM into $O(\sqrt{N})$ smaller ORAMs [45]. The partition-based framework is later used in constructing cloud-based oblivious storage [43,44]. In two independent works, Ajtai [1] and Damgård *et al.* [6] both propose statistically secure ORAM constructions. In particular, the construction by Damgård *et al.* [6] is error-free, i.e., has 0 failure probability. Fletcher *et al.* [9] propose a new technique for recursive, position map based ORAMs. This can slightly reduce the assumption on the block size to obtain the same asymptotic overhead. However, this technique requires computational assumptions instead of being statistically secure.

2 Preliminary: Tree-Based ORAM Framework

Since Circuit ORAM follows the tree-based ORAM paradigm [42], we describe the tree-based ORAM framework as background knowledge.

Notation. We use N to denote the number of (real) data blocks in ORAM, D to denote the bit-length of a block in ORAM, Z to denote the capacity of each bucket in the ORAM tree, and λ to denote the ORAM's statistical security parameter. When discussing binary trees of depth L in this paper, we say the *leaves* are at level L and the root is at level 1. For convenience in algorithm descriptions, we sometimes treat the stash as a depth-0 bucket with some capacity R that is the *imaginary parent* of the root. We also denote $\langle L \rangle := \{0, 1, 2, \ldots, L\}$, and $[a..b] := \{a, a+1, \ldots, b\}$.

2.1 Tree-based ORAM Construction

Shi *et al.* [42] proposed a new tree-based framework, which is used to build efficient ORAM schemes in many work [5, 42, 46, 47]. We now briefly review the framework.

Data structure. The server organizes *blocks* into a binary tree of height $L = \log N$; each node of the tree is a *bucket* containing Z blocks. Each block is of the form:

$$\{\mathtt{idx}||\mathtt{label}||\mathtt{data}\},$$

where `idx` is the index of a block, e.g., the (logical) address of desired block; `label` is a leaf identifier specifying the path on which the block resides; and `data` is the payload of the block, of D bits in size.

The client stores a *stash* for buffering overflowing blocks. In certain schemes such as the original binary-tree scheme [42], such a stash is not necessary. In this case, we can simply treat this as a degenerate stash of size 0.

The client also stores a *position map*, mapping a block's `idx` to a leaf `label`. As described later, position map storage can be reduced to $O(1)$ by recursively storing the position map in a smaller ORAM. These leaf labels are assigned randomly and are reassigned as blocks are accessed. If we label the leaves from 0 to $N-1$, then each label is associated with a path from the root to the corresponding leaf.

Main path invariant. Tree-based ORAMs maintain the invariant that a block marked label resides on the path from the stash (to the root) to the leaf node marked label.

Operations. Tree-based ORAMs all follow a similar recipe as shown in Figure 1. In particular, the ReadAndRm operation would read every block on the path leading to the leaf node marked `label`, and fetches and removes the block `idx` from the path.

Figure 1: `Access(op, idx, data*)`
//where op = *"read" or "write"*

```
1: $\mathtt{label} := \mathsf{PositionMap}[\mathtt{idx}]$
2: $\{\mathtt{idx}||\mathtt{label}||\mathtt{data}\} := \mathsf{ReadAndRm}(\mathtt{idx}, \mathtt{label})$
3: $\mathsf{PositionMap}[\mathtt{idx}] := \mathsf{UniformRandom}(0 \ldots N-1)$
4: If op = "read": $\mathtt{data}^* := \mathtt{data}$
5: $\mathsf{stash.add}(\{\mathtt{idx}||\mathsf{PositionMap}[\mathtt{idx}]||\mathtt{data}^*\})$
6: Evict()
7: Return data
```

Various tree-based ORAMs are differentiated by the eviction algorithm denoted Evict(). For example, the original binary-tree ORAM adopts a simple eviction algorithm engineered to make their proof easy: with each data access, two distinct buckets are chosen at random from each level to evict from. By contrast, the Path ORAM algorithm performs eviction on the read path, and the eviction strategy is aggressive: pack all blocks as close to the leaf as possible respecting the main invariant. In Path ORAM, a $O(\log N) \cdot \omega(1)$ stash is necessary to buffer overflowing blocks.

Recursion. Instead of storing the entire position map in the client's local memory, the client can store it in a smaller ORAM on the server. In particular, this position map ORAM needs to store N labels each of $\log N$ bits. We can apply this idea recursively until we get down to a constant amount of metadata, which the client could store locally.

Recursion can be done using either *uniform* or *non-uniform* block sizes. For optimizing the circuit size we will use the "big data block, little metadata block" trick described in the Path ORAM work [46]. For the position map levels of the recursion, we use a block size of $D' := \chi \log N$ for some constant χ, i.e., each little block stores χ position labels. Note that D' is not necessarily the same as the data block size D. Due to this "big data block, little metadata block" trick, our asymptotical bounds in Tables 1 and 2 have two terms, one term corresponding to the data level, and the other term corresponding to all metadata levels of the recursion. For example, in Tables 1, the binary tree ORAM's Circuit Size has two terms: the $O(D \log^2 N) \cdot \omega(1)$ part is incurred from operating on blocks (0-th level of recursion), and the $O(\log^4 N) \cdot \omega(1)$ cost is incurred due to the metadata recursion levels.

To show the tightness of certain stronger interpretations of the Goldreich-Ostrovsky lower-bound, we will use a uniform block size, and assume that blocks are of $D = N^\epsilon$ bits in size — this ensures that the recursion is of $O(1)$ depth assuming $0 < \epsilon < 1$ is a constant.

2.2 ORAM Metrics

Traditionally, ORAM schemes are ususally measured by the following metrics:

- **Bandwidth cost.** An ORAM's bandwidth cost refers to the average number of bits transferred for accessing each block of D bits.
- **Bandwidth blowup.** An ORAM's bandwidth blowup is defined as its bandwidth cost divided by D (i.e., the bit-length of a data block). Effectively, the bandwidth blowup means the multiplicative factor in bandwidth one needs to pay to get obliviousness.
- **Runtime blowup.** An ORAM's runtime blowup is defined as the runtime of the Oblivious RAM divided by the runtime of the non-oblivious RAM.

Note that in the standard RAM model with *uniform* block size, the bandwidth blowup and the runtime blowup metrics are equivalent. However, when the ORAM scheme adopts non-uniform block sizes, these two metrics may not be equal. For example, Path ORAM [46] proposes a "big data block, little metatdata block" trick, resulting in $O(\log N)$ bandwidth blowup for $\Omega(\log^2 N)$-sized blocks; however, its blowup in running time is $O(\log^2 N)$. Path ORAM achieves $O(\log N)$ blowup in running time with N^ϵ-sized blocks, where $0 < \epsilon < 1$. These above statements about Path ORAM assume $O(\log N)\omega(1)$ blocks of client storage.

As mentioned earlier, the **circuit complexity** metric for ORAMs was first raised by Wang *et al.* [47]. Below are some simple but useful observations regarding the new circuit size metric [47] for ORAM:

- If there exists an ORAM scheme with $O(f(N))$ circuit complexity (regardless of client space), then there exists an ORAM scheme with $O(D)$ of client space and $O(f(N))$ bandwidth cost.
- Conversely, for any "non-wasteful" ORAM scheme that has $\Omega(f(N))$ bandwidth cost, its circuit complexity must be at least $\Omega(f(N))$ – otherwise, some bits read from the server are not fed as inputs to the ORAM client's circuits – and hence these bits need not have been transferred.

Note that an ORAM with with $O(f(N))$ bandwidth cost does not necessarily imply an ORAM with $O(f(N))$ circuit complexity. For example, many existing hierarchical ORAM schemes [18, 19] require the computation of hash functions or PRFs on blocks of D bits. In these schemes, the circuit complexity metric would be asymptotically greater than the bandwidth cost – since for bandwidth cost, these PRF or hash computations come for free.

3 Circuit ORAM

3.1 Overview

Circuit ORAM follows the tree-based ORAM framework, by building a binary tree containing N nodes (referred to as *buckets*), where each bucket can store $Z = O(1)$ number of blocks.

Stash. As later proved in Theorem 3 and 4, with probability at least $1 - 2^{-\Omega(R)}$, the stash holds at most R blocks. We can parameterize $R = O(\log N) \cdot \omega(1)$ to obtain a failure probability negligible in N. To achieve $O(D)$ bits of client space, **this stash can be stored on the server side**, and operated on by the client in each data access obliviously. For convenience, we will often refer to the stash as being the 0-th level on the path, i.e., $\mathsf{path}[0]$.

Operations. The data access algorithm `Access` follows the same strucure as in the binary-tree ORAM [42] or Path ORAM [46] – explained in Figure 1 in Section 2. It suffices for us to describe how eviction is implemented in Circuit ORAM, which we will focus on in the remainder of this section.

Definition 1 (Legally reside) *We say that a block* B *can legally reside in* $\mathsf{path}[\ell]$ *if by placing* B *in* $\mathsf{path}[\ell]$*, the main path invariant is satisfied.*

Definition 2 (Deepness w.r.t eviction path) *For a given eviction path denoted* path*,* B_0 *is deeper than* B_1 *(with respect to* path*), if there exists some* $\mathsf{path}[\ell]$ *such that* B_0 *can legally reside in* $\mathsf{path}[\ell]$*, but* B_1 *cannot; in the case when both blocks can legally reside in the same buckets along* path*, the block with smaller index* `idx` *will be considered deeper.*

In other words, B_0 is deeper on the current eviction path than B_1 if it can legally reside nearer to the leaf along path. If two blocks have the same deepness, we use their indices `idx` to resolve ambiguity. This will be useful later in our proofs.

Our notion of deepness and the greedy eviction choice of the deepest block on a path are inspired by the novel ideas of the CLP ORAM [5] – but it will soon become apparent that we apply it in a fundamentally different manner.

3.2 Intuition

As mentioned earlier, Path ORAM's eviction algorithm is too complex when implemented as a circuit. Alternatively, we would like to have an eviction algorithm that is easy to implement as a small circuit. Based on this idea, we would like to have an eviction algorithm that ideally makes *a single scan* of the data blocks on the eviction path from the stash to leaf (and only a constant number of metadata scans).

Initial failed attempt. We explain an initial attempt that unfortunately fails to work, but will inspire our Circuit ORAM construction. As mentioned above, we will make a linear scan of the path from stash to root.

During the one-pass scan, we would like the client to "pick up" (i.e., remove from path) and hold onto one block, which can later be "dropped" somewhere further along the path. At any point of time, the client should hold onto at most one block. Further, it makes sense for the client to hold onto the currently *deepest* block when it does decide to hold a block. This way, the block in holding will have the maximum chance of being dropped later. On encountering a deeper block, the client could swap it with the one in holding.

However, a dilemma arises. How does the client decide when it should pick up a block and hold onto it? Maybe this block will never get a chance to be dropped later, in which case there will be two equally bad choices: 1) put the block into the stash – which results in rapid stash growth; and 2) go back and revisit the path to write the block back. However, doing this obliviously results in high cost.

Algorithm 1 `EvictOnceSlow(path)`
*/*A slow, non-oblivious version of our eviction algorithm, only for illustration purpose*/*

```
1: $i := L$    /* start from leaf */
2: while $i \geq 1$ do:
3:     if $\mathsf{path}[i]$ has empty slot then
4:         $(\mathsf{B}, \ell)$ := Deepest block in $\mathsf{path}[0..i-1]$ that can legally reside in $\mathsf{path}[i]$.
             /* $\mathsf{B} := \perp$ if such a block does not exist.*/
5:     end if
6:     if $\mathsf{B} \neq \perp$ then
7:         Move $\mathsf{B}$ from $\mathsf{path}[\ell]$ to $\mathsf{path}[i]$.
8:         $i := \ell$    // skip to level $\ell$
9:     else
10:        $i := i - 1$
11:    end if
12: end while
```

Remedy: lookahead mechanism with two metadata scans. The above issues result from the lack of foresight. If the client could only know when to pick up a block and place it in holding, and when to write the block back into an available slot, then these issues would have been resolved. Our idea, therefore, is to rely on two metadata scans prior to the real block scan, to compute all the information necessary for the client to develop this foresight. These metadata scans need not touch the actual blocks on the eviction path, but only metadata information such as the leaf `label` for each block, and the dummy bit indicator for each block.

3.3 Detailed Scheme Description

A slow and non-oblivious version of the eviction algorithm. To aid understanding, we first describe a slow, non-oblivious version of our eviction algorithm, `EvictOnceSlow`, as shown in Algorithm 1. This slow version only serves to illustrate the effect of the eviction algorithm, but does not describe how the algorithm can be efficiently implemented in circuit. Furthermore, this slow, non-oblivious version of our eviction algorithm gives a simpler way to reason about the stash usage of the algorithm, and hence will facilitate our proofs later. Later in this section, we describe how to implement our eviction algorithm efficiently and obliviously by making use of two metadata scans and a one real block scan; this can be readily converted into a small-sized circuit.

The `EvictOnceSlow` algorithm makes a reverse (i.e., leaf to stash) scan over the current eviction path. When it first encounters an empty slot in $\mathsf{path}[i]$, it will try to evict the deepest block B in $\mathsf{path}[0..i-1]$ to this empty slot, provided that the block B can legally reside in $\mathsf{path}[i]$. Suppose this deepest block B resides in $\mathsf{path}[\ell]$ where $\ell < i$. After relocating the block B to $\mathsf{path}[i]$, the algorithm now skips levels $\mathsf{path}[\ell+1..i-1]$, and continues its reverse scan at level ℓ instead (Line 8 in Algorithm 1). In case no block in $\mathsf{path}[0..i-1]$ can fill the empty slot in $\mathsf{path}[i]$, the scan simply continues to level $\mathsf{path}[i-1]$.

Efficient and oblivious implementation of our eviction algorithm. In Algorithm 1, Line 4 is inefficient, and Line 8 is non-oblivious. We now explain how to implement the same `EvictSlow` algorithm obliviously and efficiently, but using two metadata scans (Algorithms 2 and 3) plus a single real block scan (Algorithm 4). Since metadata is typically much smaller than real data blocks, a metadata scan is faster than a real block scan.

The two metadata scans will generate two helper data structures:

- An array $\mathsf{deepest}[1..L]$, where $\mathsf{deepest}[i] = \ell$ means that the deepest block in $\mathsf{path}[0..i-1]$ that can legally reside in $\mathsf{path}[i]$ is now in level $\ell < i$. If no block in $\mathsf{path}[0..i-1]$ can legally reside in $\mathsf{path}[i]$, then $\mathsf{deepest}[\mathsf{i}] := \perp$. In the pre-processing state, we will use one metadata scan, namely the `PrepareDeepest`

Algorithm 2 `PrepareDeepest(path)`
*/*Make a root-to-leaf linear metadata scan to prepare the* deepest *array.*
After this algorithm, deepest$[i]$ *stores the source level of the deepest block in* path$[0..i-1]$ *that can legally reside in* path$[i]$. */

```
 1: Initialize deepest := $(\perp, \perp, ..., \perp)$, src := $\perp$, goal := $-1$.
 2: if stash not empty then
          src := 0,
          goal := Deepest level that a block in path[0] can legally reside on path.
 3: end if
 4: for $i = 1$ to $L$ do:
 5:     if goal $\geq i$ then deepest[$i$] := src
 6:     end if
 7:     $\ell$ := Deepest level that a block in path[$i$] can legally reside on path.
 8:     if $\ell >$ goal then
 9:         goal := $\ell$, src := $i$
10:     end if
11: end for
```

Algorithm 3 `PrepareTarget(path)`
*/*Make a leaf-to-root linear metadata scan to prepare the* target *array. */*
After this algorithm, if target$[i] \neq \perp$*, then one block shall be moved from* path$[i]$ *to* path[target$[i]$] *in* `EvictOnceFast(path)`. */

```
 1: dest := $\perp$, src := $\perp$, target := $(\perp, \perp, \ldots, \perp)$
 2: for $i = L$ downto 0 do:
 3:     if $i$ == src then
 4:         target[$i$] := dest, dest := $\perp$, src := $\perp$
 5:     end if
 6:     if ((dest = $\perp$ and path[$i$] has empty slot) or (target[i] $\neq \perp$)) and (deepest[$i$] $\neq \perp$) then
 7:         src := deepest[i]        /* deepest is populated earlier using the PrepareDeepest algorithm.*/
 8:         dest := $i$
 9:     end if
10: end for
```

subroutine (see Algorithm 2), to populate the deepest array. This allows us to avoid Line 4 in Algorithm 1 causing an additional $\Theta(L)$ overhead.

- An array target$[0..L]$, where target$[i]$ stores which level the deepest block in path$[i]$ will be evicted to. This target array is prepopulated using a backward metadata scan as depicted in the `PrepareTarget` algorithm (see Algorithm 3).

Observe that the prepopulated target arrary basically gives a precise prescription of the client's actions (including when to pick up a block and when to drop it) during the real block scan. At this moment, the client performs a forward block scan from stash to leaf, as depicted in the `EvictOnceFast` algorithm (see Algorithm 4). The high level idea here is to "hold a block in one's hand" as one scans through the path, where the block-in-hand is denoted as hold in the algorithm. This block hold will later be written to its appropriate destination level, when the scan reaches that level.

Example. To aid understanding, a detailed example, including the PrepareDeepest, PrepareTarget, and the EvictOnceFast steps, is given in Figure 2.

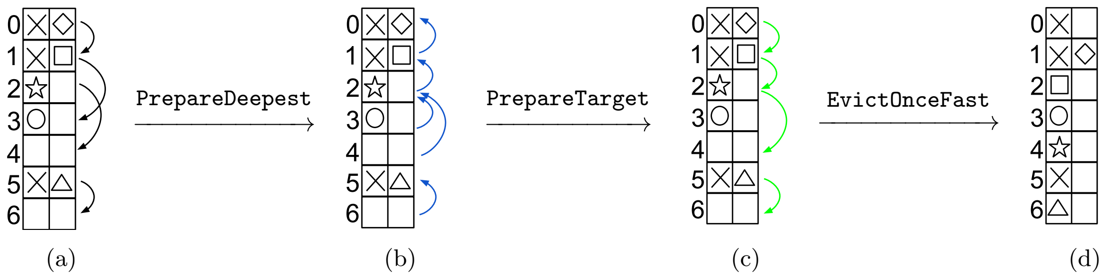


Figure 2: **An example of Circuit ORAM eviction.** An eviction path with 6 levels with stash at level 0. Bucket size and stash size are both 2. If there are two blocks in a level, $\times$ represents a block that is *not* the deepest in its level.
(a) $s \to t$: A block in $\text{path}[s]$ can legally reside in $\text{path}[t]$; but no block in $\text{path}[s]$ can legally reside in $\text{path}[t+1..L]$. Here $s < t$.
(b) $s \to t$: The deepest block in $\text{path}[0..s-1]$ that can legally reside in $\text{path}[s]$ currently resides in $\text{path}[t]$. Here $t < s$.
(c) $s \to t$: During the real block scan, the client should pick up the deepest block in $\text{path}[s]$, and drop it in $\text{path}[t]$. Here $s < t$.
(d) Blocks are evicted according to target pointers (in green).

Eviction rate and choice of eviction path. For each data access, two paths are chosen for eviction using the `EvictOnceFast` algorithm. While other approaches are conceivable, we describe two simple ways for choosing the eviction paths:

- A *random-order* eviction strategy denoted `EvictRandom()` (see Algorithm 5). The randomized strategy chooses two random paths that are non-overlapping except at the stash and the root. This means that one path is randomly chosen from each of the left and the right branches of the root.
- A *deterministic-order* strategy denoted `EvictDeterministic()` (see Algorithm 6). The deterministic-order strategy is inspired by Gentry *et al.* [11] and several subsequent works [9, 12].

Recursion. So far, we have assumed that the client stores the entire position map. Based on a standard trick [42,45,46], we can recursively store the position map on the server. In the position map recursion levels, we will use a different block size than the main data level as suggested by Stefanov *et al.* [46]. Specifically, we group c number of `labels` in one block for an appropriate constant $c > 1$. In this way, our total bandwidth cost over all recursion levels would be $O(D \log N + \log^3 N) \cdot \omega(1)$ (for negligible failure probability), assuming that the stashes reside on the server side, and the hence client only needs to hold a constant number of blocks at any time. For inverse polynomial failure probability, the total bandwidth cost is $O(D \log N + \log^3 N)$.

3.4 Formal Results

A slightly modified Circuit ORAM construction. For subtle technical reasons, in our proofs we need to make a minor modification to the main construction. Since this modification is not very interesting, we did not document it in our main scheme for clarity. The modification involves introducing an additional *partial eviction* performed on the read path, simply to fill up the hole that is newly created by the ReadAndRm operation. A partial eviction works just like a normal eviction, but works on only part of the path upto the point where the the block is removed (and for obliviousness dummy eviction operations are performed for the rest of the path). We refer the readers to Appendix A for a detailed description of the modification. This additional modified partial eviction is only necessary to to show an equivalence between a post-processed ∞-ORAM and the real ORAM (see Section 4 and Appendix A for more details).

Algorithm 4 `EvictOnceFast(path)`

```
1: Call the PrepareDeepest and PrepareTarget subroutines to pre-process arrays deepest and target.
2: hold := $\perp$, dest := $\perp$.
3: for $i = 0$ to $L$ do
4:     towrite := $\perp$
5:     if (hold $\neq \perp$) and ($i ==$ dest) then
         /* The block stored in hold will be placed in bucket path[$i$]. */
6:         towrite := hold
7:         hold := $\perp$, dest := $\perp$.
8:     end if
9:     if target[$i$] $\neq \perp$ then
10:        hold := read and remove deepest block in path[$i$]
11:        dest := target[$i$]
12:    end if
13:    Place towrite into bucket path[$i$] if towrite $\neq \perp$.
14: end for
```

Algorithm 5 `EvictRandom()`

```
1: Choose a leaf from each of the left and the right branches of the root independently, and denote the two
   corresponding (stash-to-leaf) paths by path$_0$ and path$_1$.
2: Call EvictOnceFast(path$_0$) and EvictOnceFast(path$_1$)
```

In our experiments described in Section 5, we also chose not to implement this partial eviction — this leads to slightly better empirical results. However, even if one chooses to implement this partial eviction in practice, it would only incur a small constant factor penalty.

The formal theorem statements below are for this slightly modified Circuit ORAM scheme.

Theorem 3 (Stash growth for random-order eviction.) *Let the bucket size $Z \geq 5$. Let $\mathsf{st}(\mathsf{ORAM}^Z[\mathbf{s}])$ be a random variable denoting the stash size after access sequence* $\mathbf{s}$ *for a Circuit ORAM with bucket size Z and randomized eviction. Then, for any access sequence* $\mathbf{s}$*,*

$$\Pr\left[\mathsf{st}(\mathsf{ORAM}^Z[\mathbf{s}]) > R\right] \leq 42 \cdot 0.6^R$$

where probability is taken over the ORAM algorithm's randomness.

Theorem 4 (Stash growth for deterministic-order eviction.) *Let the bucket size $Z \geq 4$. Let $\mathsf{st}(\mathsf{ORAM}^Z[\mathbf{s}])$ be a random variable denoting the stash size after access sequence* $\mathbf{s}$ *for a Circuit ORAM with bucket size Z and deterministic eviction. Then, for any access sequence* $\mathbf{s}$*,*

$$\Pr\left[\mathsf{st}(\mathsf{ORAM}^Z[\mathbf{s}]) > R\right] \leq 14 \cdot e^{-R},$$

where probability is taken over the ORAM algorithm's randomness.

While our formal proof requires $Z \geq 5$ for randomized eviction and $Z \geq 4$ for deterministic-order eviction, empirical results show that choosing $Z = 3$ for randomized eviction and $Z = 2$ for deterministic eviction would result in bounded stash size R with failure probability $2^{-\Theta(R)}$.

Theorem 5 (Circuit size under the non-uniform block size model) *Under the model with non-uniform memory word size, for any N-word RAM with word size $D = \Omega(\log^2 N)$ bits and client storage of $O(D)$ bits, there exists an ORAM construction with circuit size of $O(D \log N)\omega(1)$ achieving negligible statistical failure probability and an ORAM construction with circuit size of $O(D \log N)$ achieving inverse polynomial statistical failure probability.*

Algorithm 6 `EvictDeterministic()`

```
In timestep $t$:
1: Choose two paths, $\mathsf{path}_0$ and $\mathsf{path}_1$, corresponding to the leaves labeled with integers $\mathsf{bitrev}(2t \mod N)$ and $\mathsf{bitrev}((2t+1) \mod N)$, respectively. In the above $\mathsf{bitrev}(i)$ denotes the integer obtained by reversing the bit order of $i$ when expressed in binary.
2: Call $\mathsf{EvictOnceFast}(\mathsf{path}_0)$ and $\mathsf{EvictOnceFast}(\mathsf{path}_1)$
3: Increase $t$ by 1 for the next access.
```

Proof: For $D = \Omega(\log^2 N)$, consider Circuit ORAM with big data blocks of $O(D)$ bits, and little metadata blocks of $O(\log N)$ bits (used in the position map levels of the recursion). The rest immediately follows. ■

Theorem 6 (Runtime blowup under the uniform block size model) *Under the model with uniform memory word size, for any constant $0 < \epsilon < 1$, for a N-word RAM with word size $D = \Omega(N^\epsilon)$ bits and client storage of $O(D)$ bits, there exists an ORAM construction with runtime blowup of $O(\log N)\omega(1)$ achieving negligible statistical failure probability and an ORAM construction with runtime blowup of $O(\log N)$ achieving inverse polynomial statistical failure probability.*

Proof: When $D = \Omega(N^\epsilon)$, and using a uniform block size of D for both data and metadata blocks, we obtain $O(1)$ levels of recursion. The rest immediately follows. ■

Corollary 1 *The stronger interpretation of the Goldreich-Ostrovsky ORAM lower bound as stated in Theorem 1 is tight.*

Proof: Immediate from Theorem 6 and Theorem 1. ■

4 Proof Intuition

Our proof borrows ideas from both Path ORAM [46] and the CLP ORAM [5], but differs from both in a non-trivial manner. We informally describe the high-level proof intuition in this section, and defer the formal proofs to the Appendix.

Equivalence to post-processed ∞-ORAM. Inspired by Path ORAM [46]'s novel proof technique, we also define an ∞-ORAM analog of Circuit ORAM, where buckets are of infinite size. At the end of each data access, an admissible post-processing is defined in the same way as in the Path ORAM proof. During the post-processing, blocks in the ∞-ORAM are relocated towards the root such that there are no more than Z blocks in each bucket. Overflowing blocks are placed in the stash.

Similar to Path ORAM, we show that at the end of each data access, the real ORAM is an admissible post-processing of the ∞-ORAM; and hence to bound the stash size of the real ORAM, we only need to bound the stash size of a post-processed ∞-ORAM.

To show this equivalence to a post-processed ∞-ORAM, we give an elementary, inductive proof. While elementary, our equivalence proof is much more complicated than that of Path ORAM. The fact that the equivalence holds in our case is very non-obvious and may even be counter-intuitive at first sight. The following are essential elements in our equivalence proof:

- Extract several key properties of our algorithm — our slow, non-oblivious counterpart, namely, Algorithm 1 facilitates us in gaining these insights.
- Introduce a new notion of *episode*, which corresponds to a contiguous group of full buckets on a path; translate the admissibility of a post-processing to some requirements defined with respect to every episode on every path (Fact 2).
- Perform a rather intricate induction step that involves a thorough case-by-case analysis.

Bounding the number of blocks in each subtree in ∞-ORAM. For every subtree T containing the root, its *usage* is defined as the number of blocks in the subtree in ∞-ORAM. Based on the aforementioned equivalence, we can translate the stash bound to bounding the usage of each subtree in ∞-ORAM (Lemma 2 in Appendix A) — this step is similar to an argument used in the Path ORAM proof.

Interestingly, to bound the usage for a given subtree, we observe that the stochastic process defined on the ∞-ORAM is similar to the supermarket problem mentioned in the CLP ORAM's proof [5] — but now "cashiers" are at the tree boundary instead of in a flat level as in the CLP ORAM.

Based on CLP's terminology, we can imagine the following stochastic process.

- Every node at the boundary of the subtree is a cashier. In this intuition section, let us only consider the simple case when all cashiers correspond to internal nodes of the ORAM tree. The harder case when the subtree contains leaf nodes will be analyzed formally in Section B.

- A queue is associated with every cashier.

- Whenever a block is newly written back, it is assigned a random `label`. Suppose the path defined by `label` (i.e., the path to the leaf node marked `label`) intersects the tree boundary at some node v. Then, this block is assigned to the queue of cashier v.

- Whenever a path is selected for eviction, suppose the path exits the subtree at some boundary node v'. Then, a block will be dequeued from the queue corresponding to cashier v' — if such a block exists.

The total queue sizes of all cashiers (at the subtree boundary) would give the usage of the subtree. In other words, each arriving block will be assigned to one queue depending on its newly selected `label`. Whenever a path is selected for eviction, suppose the path exits the subtree at some boundary node v'. Circuit ORAM's eviction algorithm on the ∞-ORAM guarantees that if cashier v' has a non-empty queue, a block will be evicted out of the subtree at the tree boundary v'. See the "eviction certainty" property described in Section A.3.

For our randomized eviction strategy, whenever a new block is added to the root, two random eviction paths are chosen (that are non-overlapping except at the root and the stash). Hence, it is not difficult to see that each cashier's enqueue rate is half its dequeue rate. Therefore, for randomized eviction, the queue length distribution behaves just like a discrete-time M/M/1 queue, and the probability that the queue length exceeds x is exponentially small in x. Each cashier's queue length therefore has constant expectation and an exponentially small tail bound. At this point, we simply need to apply a measure concentration bound for the sum of the queue lengths of all cashiers. To achieve this, we need to overcome two barriers:

1. Since the queue length is an unbounded variable, standard Chernoff or Hoeffding bounds do not immediately apply. Hence, we need to prove a measure concentration bound from first principles using moment-generating functions.

2. The queue lengths of all cashiers are not independent, and hence a direct Chernoff-like bound does not apply. To tackle this issue, we show that the queue lengths of the cashiers are negatively associated.

Finally, in the formal proof, we also have to additionally deal with subtrees that include leaf nodes of the ORAM tree.

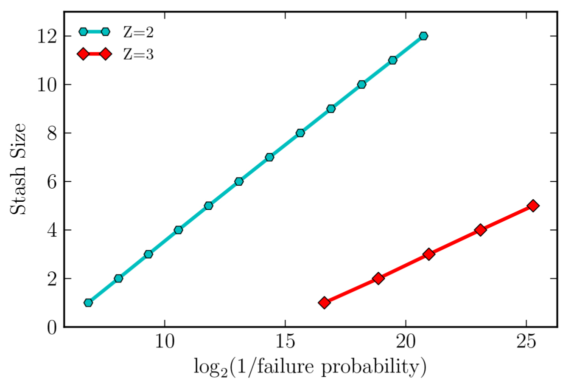
We also give an explicit proof for the deterministic-order eviction variant. The proof is similar, but requires a result on the queue length distribution of a discrete-time M/D/1 queue [27].

5 Simulation

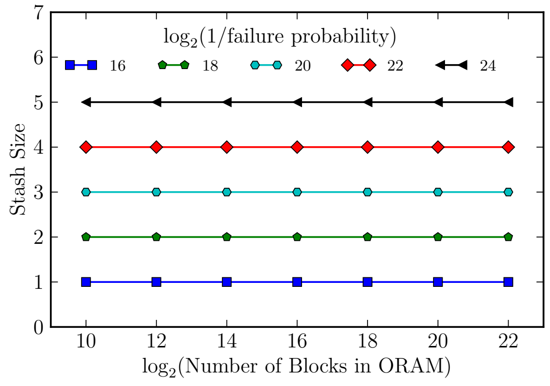
We now report our simulation results, including the stash growth and the total circuit size for moderately large datasets.

Type Of ORAM	Circuit ORAM Det.	Circuit ORAM Rand.	Path ORAM(naive) Det.	Path ORAM(naive) Rand.	Path ORAM(o-sort) Det.	Path ORAM(o-sort) Rand.	SCORAM Det.	SCORAM Rand.
Circuit size	$3.1M$	$5.5M$	$25.5M$	$148.5M$	$46.9M$	$104.1M$	$9.5M$	$11.4M$
Relative overhead	$1X$	$1.8X$	$8X$	$48X$	$15X$	$34X$	$3X$	$3.7X$
#AND gates	$0.8M$	$1.4M$	$8.5M$	$49.3M$	$14.7M$	$32.7M$	$2.6M$	$3.1M$
Relative overhead	$1X$	$1.7X$	$10.6X$	$61.6X$	$18.4X$	$40.1X$	$3.3X$	$3.9X$

Table 3: **Comparison of Circuit ORAM and variant of Path ORAM.** ORAM size = 2^{28}, block size = 32 bits, security parameter = 80. Path ORAM(o-sort) [47] uses 3 oblivious sorts. "Rand." stands for eviction with randomly chosen paths; "Det." stands for eviction with reverse-lexicographical-ordered paths, as described in earlier works [9, 11, 12].



(a) **The stash exceeds R with probability $2^{-\Theta(R)}$.** An ORAM capacity of $N = 2^{10}$ is used.



(b) **The stash size is independent of the ORAM capacity N.** A bucket size of 3 is used.

Figure 3: **Empirical evaluation of stash size.** Deterministic-order eviction is adopted.

5.1 Stash Size Distribution

We simulate Circuit ORAM for a single long run, for about 10^{10} accesses, after performing 2^{25} accesses to warm-up the ORAM data structure. It is well-known that if a stochastic process is regenerative, the time average over a single run is equivalent to the ensemble average over multiple runs (see Chapter 5 of Harchol-Balter [24]). In our experiments, we use the following request sequence: $1, 2, \ldots, N, 1, 2, \ldots, N$. Using the same argument as in Path ORAM [46], it is not hard to see that this is the worst-case access sequence.

As shown in Figure 3a, for a bucket size of 2 or 3, the stash exceeds R with probability of $2^{-\Theta(R)}$. Further, Figure 3b shows that the stash size is independent of the ORAM's capacity N. For a bucket size of 4, we never observed the stash growing beyond 5 for the first 10^{10} accesses.

5.2 Comparison of Circuit Size

In Table 3, we compare the circuit sizes of Circuit ORAM and other state-of-the-art ORAM schemes. Results in this table are obtained using the following parameters: $N = 2^{28}$, block size = 32 bits, and security parameter = 80. For Path ORAM, we consider a naive circuit implementation, and an asymptotically more efficient circuit implementation relying on three oblivious sorts [47]. For all schemes, we consider two strategies for choosing the eviction path: random-order eviction and deterministic-order eviction (based on *digit-reversed lexicographic order* [11]) The table shows that Circuit ORAM results in 8x to 48x smaller circuit size than Path ORAM, and is 3x to 3.7x better than SCORAM [47]. The relative overhead will become bigger if the block size is greater(e.g. 4K bits), because Circuit ORAM performs more meta data computation than SCORAM.

Acknowledgments

This research was funded in part by an NSF grant CNS-1314857, a subcontract from the DARPA PROCEED program, a Sloan Research Fellowship, a Google Faculty Research Award and a grant from Hong Kong RGC under the contract HKU719312E. The views and conclusions contained herein are those of the authors and should not be interpreted as representing funding agencies.

We gratefully acknowledge Bill Gasarch, Mor Harchol-Balter, Dov Gordon, Yan Huang, Jonathan Katz, Clyde Kruskal, Kartik Nayak, Charalampos Papamanthou, and abhi shelat for helpful discussions.

References

[1] M. Ajtai. Oblivious rams without cryptographic assumptions. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*, STOC '10, pages 181–190, New York, NY, USA, 2010. ACM.

[2] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In *Public-Key Cryptography–PKC 2014*, pages 131–148. Springer, 2014.

[3] P. Beame and W. Machmouchi. Making branching programs oblivious requires superlogarithmic overhead. In *Proceedings of the 2011 IEEE 26th Annual Conference on Computational Complexity*, CCC '11, pages 12–22, Washington, DC, USA, 2011. IEEE Computer Society.

[4] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. `http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf`, 2011.

[5] K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. *CoRR*, abs/1307.3699, 2013.

[6] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.

[7] D. Dubhashi and D. Ranjan. Balls and bins: a study in negative dependence. *Random Struct. Algorithms*, 13:99–124, September 1998.

[8] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[9] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, E. Stefanov, and S. Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[10] C. W. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

[11] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[12] C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345, 2014. `http://eprint.iacr.org/`.

[13] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled ram revisited. In P. Nguyen and E. Oswald, editors, *Advances in Cryptology EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422. Springer Berlin Heidelberg, 2014.

[14] C. Gentry, S. Halevi, M. Raykova, and D. Wichs. Garbled ram revisited, part i. Cryptology ePrint Archive, Report 2014/082, 2014. `http://eprint.iacr.org/`.

[15] C. Gentry, S. Halevi, M. Raykova, and D. Wichs. Outsourcing private ram computation. *IACR Cryptology ePrint Archive*, 2014:148, 2014.

[16] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[17] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *ACM symposium on Theory of computing (STOC)*, 1987.

[18] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[19] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[20] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *CCSW*, 2011.

[21] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[22] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[23] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[24] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, 2013.

[25] J. Hsu and P. Burke. Behavior of tandem buffers with geometric input and Markovian output. In *IEEE Transactions on Communications. v24*, pages 358–361, 1976.

[26] M. Keller and P. Scholl. Efficient, oblivious data structures for mpc. Cryptology ePrint Archive, Report 2014/137, 2014. `http://eprint.iacr.org/`.

[27] D. G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, pages 338–354, 1953.

[28] C. P. Kruskal, M. Snir, and A. Weiss. The distribution of waiting times in clocked multistage interconnection networks. *IEEE Trans. Computers*, 37(11):1337–1352, 1988.

[29] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[30] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient ram-model secure computation. In *IEEE S & P*. IEEE Computer Society, 2014.

[31] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. *FAST*, 2013:199–213, 2013.

[32] S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography Conference (TCC)*, 2013.

[33] S. Lu and R. Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, 2013.

[34] S. Lu and R. Ostrovsky. Garbled ram revisited, part ii. Cryptology ePrint Archive, Report 2014/083, 2014. `http://eprint.iacr.org/`.

[35] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[36] J. C. Mitchell and J. Zimmerman. Data-Oblivious Data Structures. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 25, pages 554–565, 2014.

[37] R. Ostrovsky. Efficient computation on oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1990.

[38] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *ACM Symposium on Theory of Computing (STOC)*, 1997.

[39] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[40] N. Pippenger and M. J. Fischer. Relations among complexity measures. *J. ACM*, 26(2), Apr. 1979.

[41] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

[42] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[43] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[44] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy (S & P)*, 2013.

[45] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[46] E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious ram protocol. Cryptology ePrint Archive, Report 2013/280, previous version published on CCS, 2013. `http://eprint.iacr.org/2013/280`.

[47] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[48] P. Williams and R. Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[49] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.

[50] P. Williams and R. Sion. SR-ORAM: Single round-trip oblivious ram. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[51] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.

[52] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.

[53] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[54] S. Zahur and D. Evans. Circuit structures for improving efficiency of security and privacy tools. In *S & P*, 2013.

Algorithm 7 `PartialEvictSlow(path,` ℓ^*`)`
*/*A slow, non-oblivious version of our partial eviction algorithm, only for illustration purpose. The first line is the only difference from the* `EvictSlow` *algorithm described in the main body.*/*

```
1: $i := \ell^*$    /* start from level $\ell^*$ */
2: while $i \geq 1$ do:
3:     if path[$i$] has empty slot then
4:         ($\mathsf{B}, \ell$) := deepest block and its level in path[$0 \ldots i-1$] that can reside in path[$i$] respecting
                the path invariant.    /* $\mathsf{B} := \perp$ if such a block does not exist.*/
5:     end if
6:     if $\mathsf{B} \neq \perp$ then
7:         Move $\mathsf{B}$ from path[$\ell$] to path[$i$].
8:         $i := \ell$
9:     else
10:        $i := i - 1$
11:    end if
12: end while
```

A Analyzing Stash Size via Infinity ORAM

A.1 A Slight Variant of Circuit ORAM

We would like to prove bounds for stash usage in the main scheme described in Section 3. For technical reasons, we will prove bounds for a slight variant of the scheme that performs some additional evictions. Recall that we use *hole* to mean an available slot in a bucket that was previously occupied by another block, or the only available slot in a bucket.

In the main scheme described in Section 3, we do not perform eviction on the read path, but perform ν number of evictions for paths chosen in a deterministic order. In this new variant, however, we will also perform a "partial eviction" on the read path. The purpose of this is to fill the hole created by reading and removing a block, so that we can prove a strong equivalence property to relate the real ORAM to a post-processed ∞-ORAM, whose purpose is solely for analysis. The eviction is partial, because it is only performed on the segment $\mathsf{path}[0 \ldots \ell^*]$, i.e., from the stash to the level ℓ^* where a block is fetched and removed.

For simplicity, in Algorithm 7, we describe a slow, non-oblivious version of the partial eviction. Using the same techniques described in Section 3, this partial eviction can be done by calling the `PrepareDeepest` and `PrepareTarget` subroutines, followed by the `EvictFast` on the segment $\mathsf{path}[0 \ldots \ell^*]$. To hide where the level ℓ^* is, one needs to perform the corresponding dummy operations on the segment $\mathsf{path}[\ell^* + 1 \ldots L]$.

One could also implement this partial eviction in our real scheme. This would add a small constant factor to our overhead. For our real implementation, we chose not to implement this partial eviction as an empirical optimization.

A.2 Infinity ORAM

Recall that we treat the stash as a bucket $\mathsf{path}[0]$ that is the imaginary "parent" of the root $\mathsf{path}[1]$.

Definition 3 (∞-ORAM for Circuit ORAM) *∞-ORAM is defined in the same way as our ORAM, except that each bucket has infinite capacity.*

We stress that ∞-ORAM is only a construct used in our proof to analyze stash usage – in fact, ∞-ORAM is not oblivious, since the current bucket load leaks information. To distinguish between the buckets along path in real ORAM and ∞-ORAM, we use path_R and path_∞; the subscripts are dropped when the description applies to both ORAMs.

Post-processed ∞-ORAM. At the end of each time step, let S denote the state of the ∞-ORAM, let S' denote the state of a real ORAM with bucket capacity Z. We say that S' is an *admissible* post-processing of S, *iff* the following conditions hold:

1. For every block B residing in some node bucket in S, B must reside in an ancestor of bucket or bucket itself in S'.
2. If a block resides in $\mathsf{path}_R[\ell']$ in S', and resides in $\mathsf{path}_\infty[\ell]$ in S, where $\ell > \ell'$, then it must be the case that all buckets in $\mathsf{path}_R[\ell' + 1 \dots \ell]$ are full in S'.

In the real-world algorithm in Section 3.3, we allow an arbitrary choice when two blocks can be legally evicted to the same depth along the eviction path. For the purpose of the proof, we assume that the tie is resolved by choosing the block with the smaller block index. This rule is applied to both the real ORAM and the ∞-ORAM. By resolving the ambiguity, we can keep the real ORAM and the ∞-ORAM synchronized, such that we can prove a strong equivalence condition between the two.

Lemma 1 (Strong equivalence) *After any request sequence* $\mathbf{s}$*, and randomness sequence* $\mathbf{r}$*, the real ORAM is an admissible post-processing of the* ∞*-ORAM.*

In particular, this implies that the stash usage in a post-processed ∞*-ORAM is exactly the same as the stash usage in the real ORAM.*

Below, we will prove Lemma 1 by induction.

Fact 1 *(Base case.) At time step* 0*, the strong equivalence condition (Lemma 1) holds between an admissible post-processed* ∞*-ORAM and the real ORAM.*

The above fact can be trivially observed, since at time $t = 0$, there is no block in either the ∞-ORAM or the real ORAM.

We assume that the ORAM proceeds in *steps.* In each step, one of the following things happen: 1) eviction is performed on a path selected in a deterministic order; and 2) a block is read and removed from a read-path, and a partial eviction is performed. These operations are applied to both the real ORAM and the ∞-ORAM.

Lemma 2 *(Induction step.) If for any* $t < k$*, the strong equivalence condition (Lemma 1) holds, then the condition holds for time step* $t = k$*.*

The remainder of this section will focus on the proof of the induction step.

A.3 Useful Properties of Our Eviction Algorithm

Observe that our eviction algorithm in Section 3.3 or partial eviction algorithm has the following properties for both the real ORAM and ∞-ORAM.

- **Eviction certainty.** For full-path eviction: if level ℓ is *non-full*[2], and there exists a block in $\mathsf{path}[0 \dots \ell - 1]$ that can legally reside in $\mathsf{path}[\ell]$ or its descendant, then a block will move from $\mathsf{path}[0 \dots \ell - 1]$ into $\mathsf{path}[\ell \dots L]$.

  For partial eviction, this holds for the levels $\ell \leq \ell^*$ where ℓ^* is the level of the block being read and removed.

- **Choice of block.** For full-path eviction: for a *non-full* level $\ell \geq 1$, suppose that there exists a block in $\mathsf{path}[0 \dots \ell - 1]$ that can legally reside in $\mathsf{path}[\ell \dots L]$. Then, the block in $\mathsf{path}[0 \dots \ell - 1]$ that can be moved deepest along path (where ties are resolved by choosing the block with the smallest index) will be chosen to be moved to $\mathsf{path}[\ell \dots L]$.

  For partial eviction, this holds for the levels $\ell \leq \ell^*$ where ℓ^* is the level of the block being read and removed.

[2]We assume that every level is non-full in the ∞-ORAM.

- **Mutually exclusive.** At most one block from $\mathsf{path}[0 \ldots \ell-1]$ will be evicted into $\mathsf{path}[\ell \ldots L]$.
- **Filling a hole created during read-and-remove or eviction.** Suppose a block is removed from $\mathsf{path}[\ell]$ to create a "hole", either in the case $\ell = \ell^*$ due to a read-and-remove, or in Line 7 of Algorithm 1 or 7. Suppose at this moment, there exists a block in $\mathsf{path}[0 \ldots \ell-1]$ that can legally reside in $\mathsf{path}[\ell]$. Then, in the round (in Algorithm 1 or 7) where i is set to ℓ, this hole will be filled. Notice that other available slots in level ℓ will not be filled.

A.4 Proof of Lemma 2

Recall that in the induction step, there is an eviction path; in the case that the eviction is partial, eviction is performed from level 0 (the stash) to some level ℓ^*, at which level a block is read and removed.

To avoid ambiguity, in our induction proof, we will use $\mathsf{epath} := \mathsf{epath}[0..L]$ (or $\mathsf{epath} := \mathsf{epath}[0..\ell^*]$ in the case of a partial eviction) to denote the eviction path in the k-th step, and use path to denote any arbitrary path.

We use the notation path_R^k to denote a path in the real ORAM at the end of the k-th time step, and use path_∞^k to denote a path in the ∞-ORAM at the end of the k-th step. Sometimes we omit the superscript or the subscript if it does not matter which ORAM or what time step we refer to.

We partition each path $\mathsf{path}_R[0..L]$ in the real ORAM into *episodes.*

Definition 4 (Episode) *For $0 \le a \le b \le L$, we say that $\mathsf{path}_R[a..b]$ is an episode in the real ORAM, iff the following holds:*

1. *Bucket $\mathsf{path}_R[a]$ is not full (recall that the stash $\mathsf{path}_R[0]$ is by default not full), while for all other levels i in the episode, the bucket $\mathsf{path}_R[i]$ is full.*
2. *Either $b = L$, or $\mathsf{path}_R[b+1]$ is not full in the real ORAM; in the latter case, we say that b is an episode* boundary.

Notice that the smallest possible episode contains a single non-full level. A path is partitioned into contiguous groups of buckets, where each group of buckets is indexed by an episode. For convenience, we will number the episodes on a path from the stash to the leaf.

Fact 2 (Characterizing admissibility via episodes) *A real ORAM is an admissible post-processing of some ∞-ORAM* iff *the following conditions hold.*

(a) As before, every block residing in some bucket of ∞-ORAM must reside in an ancestor (not necessarily proper) bucket in the real ORAM.

(b) For every path and every episode $\mathsf{path}_R[a..b]$, the blocks in $\mathsf{path}_\infty[a..b]$ are contained in $\mathsf{path}_R[a..b]$ in the real ORAM.

Remark. Observe that Fact 2(a) is always satisfied, as eviction is carried out more aggressively in ∞-ORAM because there is no bucket capacity constraint. Hence, in our proofs, we mainly concentrate on proving the condition Fact 2(b).

Fact 3 *Suppose a real ORAM is an admissible post-processing of some ∞-ORAM. Then, for any episode $\mathsf{path}_R[a..b]$, and $\ell \in [a..b]$, the blocks in $\mathsf{path}_\infty[a..\ell]$ are contained in $\mathsf{path}_R[a..\ell]$.*

Fact 3 is obvious by the definition of post-processing and by Fact 2(b).

Recall that if $\mathsf{path}_R[a..\ell]$ and $\mathsf{path}_R[\ell+1..b]$ are adjacent episodes, then ℓ is called an *episode boundary.*

Lemma 3 *Suppose a real ORAM is an admissible post-processing of some ∞-ORAM. Then, for any path, for any episode boundary ℓ (defined with respect to $\mathsf{path}_R[0..L]$), the blocks in $\mathsf{path}[0..\ell]$ that can legally reside in $\mathsf{path}[\ell+1]$ are exactly the same in the real ORAM and ∞-ORAM.*

Proof: Suppose a real ORAM is an admissible post-processing of some ∞-ORAM. If a block resides in $\mathsf{path}_\infty[0..\ell]$, then it must reside in $\mathsf{path}_R[0..\ell]$ by definition of post-processing.

For the other direction, we prove by contradiction. Suppose a block B resides in $\mathsf{path}_R[0..\ell]$ and can legally reside somewhere in $\mathsf{path}[\ell+1..L]$, but block B is not in $\mathsf{path}_\infty[0..\ell]$. Then, by definition of post-processing, it must be in $\mathsf{path}_\infty[\ell+1..L]$. Now, by Fact 2(b), since ℓ is an episode boundary, block B must be in $\mathsf{path}_R[\ell+1..L]$, leading to a contradiction. ■

Lemma 4 *Let $\mathsf{path}_R^{k-1}[a..b]$ be an episode at the end of the $(k-1)$-th step, and let $\ell \in [a..b]$. If a block B is in $\mathsf{path}_\infty^{k-1}[a..\ell]$ and remains in $\mathsf{path}_\infty^{k}[a..\ell]$, then block B is in $\mathsf{path}_R^{k}[a..\ell]$.*

Proof: If $\mathsf{path}[a..\ell]$ does not intersect with the eviction path epath in the k-th step, then the result is trivial, because no block is moved into or out of $\mathsf{path}[a..\ell]$ in both the real ORAM or ∞-ORAM in the k-th step; the induction hypothesis implies the result. Otherwise, $\mathsf{path}[a..\ell]$ intersects with epath on levels $[a..\ell']$, where $\ell' \in [a..\ell]$. We consider two cases.
Case (1): a block B is in $\mathsf{path}_\infty^{k-1}[\ell'+1..\ell]$. By the induction hypothesis, B is in $\mathsf{path}_R^{k-1}[a..\ell]$, and must still be in $\mathsf{path}_R^{k}[a..\ell]$, since it cannot have been moved elsewhere in the real ORAM.
Case (2): a block B is in $\mathsf{path}_\infty^{k-1}[a..\ell']$. Observe that in the k-th step, at most one block from $\mathsf{path}_\infty^{k-1}[a..\ell']$ will be moved to $\mathsf{epath}[\ell'+1]$ or its descendants. By Fact 3 and the induction hypothesis, all blocks in $\mathsf{path}_\infty^{k-1}[a..\ell']$ must be in $\mathsf{path}_R^{k-1}[a..\ell']$. If B stays in $\mathsf{path}_\infty^{k}[a..\ell']$ at the end of the k-th step, it means either B cannot legally reside in $\mathsf{epath}[\ell'+1]$ on the eviction path, or B is not the deepest block in $\mathsf{path}_\infty^{k-1}[0..\ell']$ (with respect to epath). If the former happens, B must still be in $\mathsf{path}_R^{k}[a..\ell']$ at the end of the k-th step; if the latter happens, by Lemma 3 we know that B will not be the deepest in $\mathsf{path}_R^{k-1}[0..\ell']$ (with respect to epath), and hence will also stay in $\mathsf{path}_R^{k}[a..\ell']$. Note that this implicitly relies on using a block's `idx` to resolve any ambiguity in the notion of deepness.

■

Lemma 5 *Let $\mathsf{path}_R^{k-1}[a..b]$ be an episode at the end of time step $(k-1)$. A block B newly enters $\mathsf{path}_\infty^{k}[a..b]$ in the k-th step* iff *it also does so in $\mathsf{path}_R^{k}[a..b]$.*

Proof: Not hard to see given Lemma 3, the definition of an episode, and by the properties of our eviction algorithm. ■

Lemma 6 *Let $\mathsf{path}_R^{k-1}[a..b]$ be an episode at the end of the $(k-1)$-st step. Then, all blocks in $\mathsf{path}_\infty^{k}[a..b]$ must be in $\mathsf{path}_R^{k}[a..b]$.*

Proof: If a block in $\mathsf{path}_\infty^{k-1}[a..b]$ remains in $\mathsf{path}_\infty^{k}[a..b]$, then it must be in $\mathsf{path}_R^{k}[a..b]$. At most one new block B could enter $\mathsf{path}_\infty^{k}[a..b]$; in this case, B must also enter $\mathsf{path}_R^{k}[a..b]$ by Lemma 5. ■

Fact 4 *Suppose that the induction hypothesis holds at the end of $(k-1)$-th step, and $\mathsf{path}_R^{k-1}[a..b]$ is an episode. Then, $\mathsf{path}_R^{k}[a+1..b]$ has at most one hole in it.*

Definition 5 (Sub-episode after k-th step) *Let $\mathsf{path}_R^{k-1}[a..b]$ be an episode. Suppose after the k-th step, a hole is created in bucket $\mathsf{path}_R^{k}[h]$, where $h \in [a+1..b]$. Then, we say that episode $\mathsf{path}_R^{k-1}[a..b]$ is split into two* **sub-episodes** *$\mathsf{path}_R^{k}[a..h-1]$ and $\mathsf{path}_R^{k}[h..b]$ after the k-th step. If no new hole is created in $[a+1..b]$, then there is a single sub-episode $\mathsf{path}_R^{k}[a..b]$ at the end of the k-th step.*

Remark. Observe that the sub-episode $\mathsf{path}_R^{k}[a..b]$ might not be an episode after the k-th step, because in the k-th step, a block might fill the only available slot in $\mathsf{path}_R^{k}[a]$, causing the sub-episode to be merged with the one preceding it.

Fact 5 *For any path, each sub-episode as defined above must be contained entirely within an episode at the end of the k-th step, or disjoint with an episode at the end of the k-th step. In other words, each episode at the end of the k-th step can be partitioned into one or more (actually at most two) sub-episodes.*

Due to Fact 5, in order for us to prove the induction step of Lemma 2, it suffices to prove the following condition in Lemma 7 that is stronger than the condition in Fact 2(b).

Lemma 7 *Suppose* $\mathsf{path}^k_R[a..b]$ *is a sub-episode after the k-th step. Then, every block in* $\mathsf{path}^k_\infty[a..b]$ *must be in* $\mathsf{path}^k_R[a..b]$ *as well.*

Proof: We consider cases according to how the sub-episode $\mathsf{path}^k_R[a..b]$ is formed.
Case 1: $\mathsf{path}^{k-1}_R[a..b]$ is an episode at the end of the $(k-1)$-st step. This case follows from Lemma 6.

Otherwise, the sub-episode is the result of splitting an episode during the k-th step; there are two subcases.

Case 2(a): $\mathsf{path}^k_R[a..b]$ is a sub-episode resulting from splitting an episode $\mathsf{path}^{k-1}_R[\widehat{a}..b]$, for some $\widehat{a} < a$; during the k-th step, one block is removed from the full bucket $\mathsf{path}^{k-1}_R[a]$ to create a hole in $\mathsf{path}^k_R[a]$, either due to a read-and-remove or an eviction. In either case, $\mathsf{path}[0..a]$ is on the eviction path (partial for a read-and-remove) epath.

Suppose, for contradiction's sake, that there is some block B in $\mathsf{path}^k_\infty[a..b]$ that is not in $\mathsf{path}^k_R[a..b]$.

Then, since Fact 2(a) always holds, block B must be in $\mathsf{path}^k_R[0..a-1]$. Since block B is in $\mathsf{path}^k_\infty[a..b]$, it can legally reside in $\mathsf{path}[a]$. By the eviction certainty property in Section A.3, one block would have been moved from $\mathsf{path}^{k-1}_R[0..a-1]$ to fill the hole in $\mathsf{path}^k_R[a]$, causing a contradiction.

Case 2(b): $\mathsf{path}^k_R[a..b]$ is a sub-episode resulting from splitting an episode $\mathsf{path}^{k-1}_R[a..\widehat{b}]$, for some $\widehat{b} > b$; during the k-th step, one block is removed from the full bucket $\mathsf{path}^{k-1}_R[b+1]$ to create a hole in $\mathsf{path}^k_R[b+1]$, either due to a read-and-remove or an eviction. In either case, $\mathsf{path}[0..b+1]$ is on the eviction path (partial for a read-and-remove) epath.

Suppose a block B is in $\mathsf{path}^k_\infty[a..b]$. Then, either B is also in $\mathsf{path}^{k-1}_\infty[a..b]$, or newly enters $\mathsf{path}^k_\infty[a..b]$.

In the first case, Lemma 4 states that block B is in $\mathsf{path}^k_R[a..b]$. In the second case, Lemma 5 states that block B newly enters $\mathsf{path}^k_R[a..\widehat{b}]$; however, since there is a hole in $\mathsf{path}^k_R[b+1]$, this newly entering block B could not have moved beyond level b, and so must stay in $\mathsf{path}^k_R[a..b]$. ■

B Analyzing Stash Usage of Infinity ORAM

In Section A, we show an equivalence relationship (Lemma 1) between the real ORAM and a post-processed ∞-ORAM. Following the proof strategy and the notation as in [46], we use $\textsc{oram}^Z$ to denote a real ORAM with bucket size Z, and $\textsc{oram}^\infty$ to denote ∞-ORAM. Given an access sequence $\mathbf{s}$, the configurations of the ORAMs are denoted by $\textsc{oram}^Z[\mathbf{s}]$ and $\textsc{oram}^\infty[\mathbf{s}]$. The stash usage of the real ORAM is denoted as $\mathsf{st}(\textsc{oram}^Z[\mathbf{s}])$, and the stash usage of a post-processed ∞-ORAM to an ORAM with bucket size Z is denoted as $\mathsf{st}^Z(\textsc{oram}^\infty[\mathbf{s}])$. Then, the strong equivalence lemma (Lemma 1) states that $\mathsf{st}(\textsc{oram}^Z[\mathbf{s}]) = \mathsf{st}^Z(\textsc{oram}^\infty[\mathbf{s}])$.

Given a subtree T of the ∞-ORAM tree, the number of blocks contained in the buckets of subtree T after ∞-ORAM processes the request sequence $\mathbf{s}$ is denoted as $\mathsf{u}^T(\textsc{oram}^\infty[\mathbf{s}])$. An observation made in [46] is that for any $R \geq 0$, $\mathsf{st}^Z(\textsc{oram}^\infty[\mathbf{s}]) > R$ *iff* there exists a subtree T such that $\mathsf{u}^T(\textsc{oram}^\infty[\mathbf{s}]) > n(T) \cdot Z + R$, where $n(T)$ is the number of buckets in subtree T.

The following equations summarize the above discussion.

$$\begin{aligned} & \Pr[\mathsf{st}(\textsc{oram}^Z[\mathbf{s}]) > R] \\ = \; & \Pr[\mathsf{st}^Z(\textsc{oram}^\infty[\mathbf{s}]) > R] \\ = \; & \Pr[\exists T \; \mathsf{u}^T(\textsc{oram}^\infty[\mathbf{s}]) > n(T)Z + R] \\ \leq \; & \sum_T \Pr[\mathsf{u}^T(\textsc{oram}^\infty[\mathbf{s}]) > n(T)Z + R], \end{aligned}$$

where T ranges over all subtrees containing the root, and the inequality follows from the union bound.

Since the number of ordered binary trees of size n is equal to the Catalan number C_n, which is $\leq 4^n$,

$$\Pr[\mathsf{st}(\textsc{oram}^Z[\mathbf{s}]) > R] \leq \sum_{n \geq 1} 4^n \max_{T : n(T) = n} \Pr[\mathsf{u}^T(\textsc{oram}^\infty[\mathbf{s}]) > nZ + R]. \tag{1}$$

Fixing some subtree T with n nodes (buckets), we next give a uniform upper bound for $\Pr[\mathsf{u}^T(\textsc{oram}^\infty[\mathbf{s}]) > nZ + R]$ in terms of n, Z and R. Although we make measure concentration argument similar to [46], the underlying random processes are actually very different. For instance, the blocks in [46] are evicted independently from one another, while the blocks in this case are dependent upon one another.

B.1 Analyzing Usage of Subtree

We consider the usage $\mathsf{u}^T(\mathsf{ORAM}^\infty[\mathbf{s}])$ of a subtree T in ORAM^∞ after a sequence $\mathbf{s}$ of τ accesses, starting from an initially empty ORAM. Recall that N is the number of distinct blocks; we assume that N is a power of 2, and the ORAM binary tree has N leaves. We shall prove the following lemma in this subsection.

Lemma 8 (Subtree usage with random eviction) *Suppose* $\mathbf{s}$ *is an access sequence, and* T *is a subtree with* $n = n(T)$ *nodes containing the root of the binary ORAM tree. Then, for* $Z = 5$*, for any* $R > 0$*,*
$\Pr[\mathsf{u}^T(\mathsf{ORAM}^\infty[\mathbf{s}]) > n \cdot Z + R] \leq 3 \cdot \frac{1}{4^n} \cdot (0.93312)^n \cdot (0.6)^{-R}$.

We next define some notations relating to the subtree T.

Definition 6 (Exit node) *For a given path* P *from the root to some leaf node, suppose that some node* v *is the first node of the path* P *that is not part of* T*, then we refer to node* v *as the exit node, denoted* $v := \mathsf{exit}(P, T)$*. If the whole path* P *is contained in* T*, then the exit node* $\mathsf{exit}(P, T)$ *is* null.

Let F be the set of nodes in T that are also leaves of the ORAM binary tree; we denote $l := |F|$. We augment the tree T by adding nodes to form $\widehat{T}$ in the following way. If a node in T has any child node v that is not in T, then node v will be added to $\widehat{T}$. The added nodes in $\widehat{T}$ are referred to as exit nodes, denoted by E; the leaves of $\widehat{T}$ are denoted by $\widehat{E} = E \cup F$. Observe that if T contains n nodes and $|F| = l$, then $|\widehat{E}| = n - l + 1$. Define $\widehat{E}_L$ and $\widehat{E}_R$ to be the nodes of $\widehat{E}$ in the left and the right branch of the root respectively; hence, $\widehat{E}$ is the union of the disjoint sets $\widehat{E}_L$ and $\widehat{E}_R$.

For each node $u \in \widehat{E}$, define p_u to be the fraction of leaves in the original ORAM tree that are descendants of u. Observe that $p_u \leq \frac{1}{2}$. In particular, if $u \in F$, $p_u = \frac{1}{N}$; moreover, $\sum_{u \in \widehat{E}} p_u = 1$.

We summarize the notations we use in the following table.

Variable	Meaning
T	a subtree rooted at the root of the ORAM_∞ binary tree
$\widehat{T}$	augmented tree by including every child (if any) of every node in T
F	nodes of a subtree T that are leaves to the ORAM binary tree
E	set of exit nodes of a subtree T
$\widehat{E} := E \cup F$	set of all leaves of $\widehat{T}$
Z	capacity of each bucket

Defining usage for nodes in $\widehat{E}$. Consider the blocks that reside in tree T after the access sequence $\mathbf{s}$ is processed for i steps. For each node $u \in \widehat{E}$, define X_u^i to be the number of blocks in T after step i that have labels corresponding to root-to-leaf paths that intersect node u. When the context is clear, we may simplify the notation by dropping the superscript i. Hence, after the access sequence is performed, it follows that $\mathsf{u}^T(\mathsf{ORAM}^\infty[\mathbf{s}]) = \sum_{u \in \widehat{E}} X_u$. Even though the X_u's are not independent, we shall show that they are negative associated in order to prove large deviation bounds.

Stochastically dominating assumptions. To simplify the proof, we make worst case scenario assumptions in the sense that under these assumptions, the random variable for the usage of the subtree T stochastically dominates the original random variable without these assumptions.

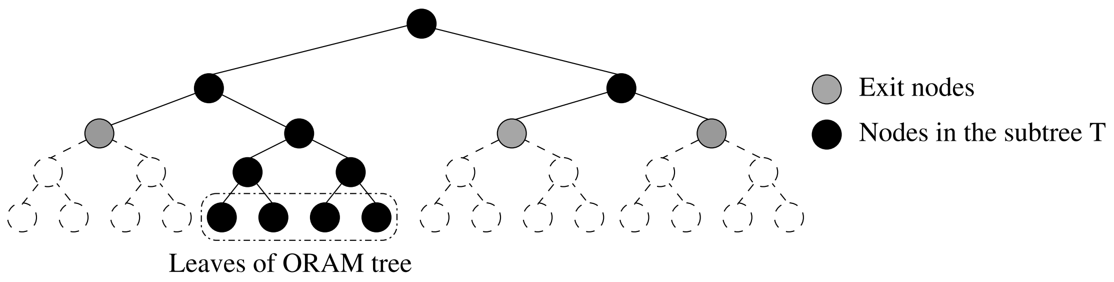


Figure 4: A subtree containing some leaves of the original ORAM binary tree, augmented with the exit nodes. This diagram is also used in the paper [46].

1. For a block whose label corresponds to a root-to-leaf path that intersects an exit node $u \in E$, the block is not removed from the subtree T when a read request is made for that block (even though a new block with a new label is added to the stash); in other words, we assume that such a block can only leave the subtree T by eviction through the exit node u. For a block whose label corresponds to some leaf in F, the block is removed as usual when a read request is made for it.
2. No partial eviction is performed on the reading path.

Defining the random process corresponding to the ORAM operations. Consider an access sequence $\mathbf{s} := \{s_i : i \in [\tau]\}$, where s_i is the identity of the block requested at step i. For every $i \in [\tau]$, we define random variables that capture the randomness used in the ORAM operations.

1. **Read.** After the desired block is read, a fresh random leaf is assigned as its label, and the block is put at the root bucket. Since we are concerned only about which node in $\widehat{E}$ is the ancestor of the new label, for each $u \in \widehat{E}$, we define a random variable $R_u^i \in \{0, 1\}$ such that $\sum_{u \in \widehat{E}} R_u^i = 1$ and $\Pr[R_u^i = 1] = p_u$.
2. **Evict.** A random eviction path is picked independently for each of the left and the right branch of the root. This corresponds to sampling random variables $S_u^i \in \{-1, 0\}$ for $u \in \widehat{E}$ such that $\sum_{u \in \widehat{E}_L} S_u^i = \sum_{u \in \widehat{E}_L} S_u^i = 1$ and $\Pr[S_u^i = -1] = 2p_u$; moreover, the randomness for the left branch is independent of that for the right branch.

Observe that given an access sequence $\mathbf{s}$, for each $u \in \widehat{E}$, the random variables $\{(R_u^i, S_u^i)\}_{i \in [\tau]}$ contain all the information to determine X_u at the end of the process. In particular, we describe the cases whether u is in F or E.

1. Case $u \in F$. In this case, u has no descendant and no block can be evicted from u. Hence, the S_u^i's are irrelevant. Therefore, X_u corresponds to the blocks that are last assigned to u. In the worst case where all N blocks appear in the access sequence, X_u has the same distribution as the sum of N independent $\{0, 1\}$-random variables, each of which has expectation $\frac{1}{N}$.
2. Case $u \in E$. In this case, u is an exit node. Recall we assume that the blocks that can be evicted through u can leave subtree T only by eviction through u. Hence, at every step i, if $R_u^i = 1$, then X_u is increased by 1; if $S_u^i = -1$ and X_u is non-zero (possibly just increased because $R_u^i = 1$), then X_u is decreased by 1.
   Observe that this defines a Markov process with non-negative integral states having arrival rate $\alpha = p_u$ and departure rate $\beta = 2p_u$. From the result by Hsu and Burke [25], this process has stationary distribution $\pi_j = (1 - q_u)^j q_u$, where $q_u := \frac{\beta - \alpha}{\beta(1-\alpha)} = \frac{1}{2(1-p_u)} \geq \frac{1}{2}$.
   Instead of starting at $X_u^0 = 0$, if we consider the stochastically dominating process such that X_u^0 has the stationary distribution (with independent randomness), then at the end of the access sequence, the corresponding random variable X_u^∞ stochastically dominates X_u, and also has the stationary distribution.

Moment generating functions. It is standard technique to consider moment generating functions in large deviation bounds. The following upper bounds on the moment generating functions will be used later.

Lemma 9 (Upper Bounds on Moment Generating Functions) *Suppose X_u's are defined as above.*

1. *For $u \in F$, any real t, $\mathbb{E}[e^{tX_u}] \leq \exp(e^t - 1)$.*

2. For $u \in E$, for $0 < t \leq \ln 2$, $\mathbb{E}[e^{tX_u}] \leq \frac{1}{2-e^t}$.
Moreover, for $\prod_{u \in \widehat{E}} \mathbb{E}[e^{tX_u}] \leq (\frac{1}{2-e^t})^{n+1}$, where n is the number of nodes in the subtree T.

Proof: For $u \in F$, observe that in our construction, we can assume that X_u is the sum of N independent $\{0,1\}$-random variables, each of which has mean $\frac{1}{N}$. Hence, for any real t, we have $\mathbb{E}[e^{tX_u}] = ((1 - \frac{1}{N}) + \frac{1}{N}e^t)^N \leq \exp(e^t - 1)$.

For $u \in E$, as described in our construction, X_u is stochastically dominated by X_u^∞, which has stationary distribution $\pi_j = (1-q_u)^j q_u$, for $j \geq 0$, where $q_u = \frac{1}{2(1-p_u)}$. Hence, for $0 \leq t \leq \ln 2 \leq \ln \frac{1}{1-q_u}$, we have

$\mathbb{E}[e^{tX_u}] \leq \mathbb{E}[e^{tX_u^\infty}] = \frac{q_u}{1-(1-q_u)e^t} \leq \frac{1}{2-e^t}$, because the maximum is attained when q_u approaches $\frac{1}{2}$.

As for the product, observe that the number of terms is at most $|\widehat{E}| = n + 1 - l \leq n+1$. Since for $t \in [0, \ln 2]$, $\exp(e^t - 1) \leq \frac{1}{2-e^t}$, the result follows.

■

Negative association. We remark that X_u is a non-decreasing function of $\{(R_u^i, S_u^i)\}_{i \in [\tau]}$, which means that if a single variable is increased (either from 0 to 1 or from -1 to 0), the function does not decrease. Hence, from [7, Proposition 7(2)], in order to show that the random variables X_u's are negative associated, it suffices to show that the whole collection $\{(R_u^i, S_u^i)\}_{u, \widehat{E}, i \in [\tau]}$ are negatively associated. We first recall the definition of negative association.

Definition 7 (Negative association [7]) *A set of random variables $X_1, X_2, \ldots, X_k$ are negatively associated, if for every two disjoint index sets, $I, J \subseteq [k]$, for all non-decreasing functions $f : \mathbb{R}^{|I|} \to \mathbb{R}$ and $g : \mathbb{R}^{|J|} \to \mathbb{R}$, $\mathbb{E}[f(X_i, i \in I) g(X_j, j \in J)] \leq \mathbb{E}[f(X_i, i \in I)]\mathbb{E}[g(X_j, j \in J)]$.*

Observe that if each of two mutually independent collections of random variables are negatively associated, then so are their union [7, Proposition 7(1)]. Hence, it suffices to show that each collection of correlated random variables in our construction are negatively associated.

The following lemma is the generalization of the Zero-One Lemma [7, Lemma 8]. As mentioned in [7], the idea of the proof is due to Colin McDiarmid.

Lemma 10 (At Most One Non-Zero Random Variable) *Suppose $\mathbf{X}$ is a collection of random variables either all having non-negative support or all having non-positive support. Moreover, with probability 1, at most one of them can be non-zero. Then, the collection $\mathbf{X}$ are negatively associated.*

Proof: Suppose $\mathbf{X}_I$ and $\mathbf{X}_J$ are disjoint subsets of the collection, and f and g are non-decreasing functions on $\mathbf{X}_I$ and $\mathbf{X}_J$ respectively.

If the random variables have non-negative support, then both f and g attains their minimum at $\mathbf{0}$; otherwise, all random variables have non-positive support, and both f and g attains their maximum at $\mathbf{0}$. Hence, either both $\mathbb{E}[f(\mathbf{X}_I) - f(\mathbf{0})]$ and $\mathbb{E}[g(\mathbf{X}_J) - g(\mathbf{0})]$ are non-negative, or both are non-positive. In any case, $0 \leq \mathbb{E}[f(\mathbf{X}_I) - f(\mathbf{0})] \cdot \mathbb{E}[g(\mathbf{X}_J) - g(\mathbf{0})]$.

On the other hand, with probability 1, at most one of $\mathbf{X}_I$ and $\mathbf{X}_J$ can be non-zero. Hence, with probability 1, $(f(\mathbf{X}_I) - f(\mathbf{0})) \cdot (g(\mathbf{X}_J) - g(\mathbf{0})) = 0$, which implies that $\mathbb{E}[(f(\mathbf{X}_I) - f(\mathbf{0})) \cdot (g(\mathbf{X}_J) - g(\mathbf{0}))] = 0$.

Therefore, we have $\mathbb{E}[(f(\mathbf{X}_I) - f(\mathbf{0})) \cdot (g(\mathbf{X}_J) - g(\mathbf{0}))] \leq \mathbb{E}[f(\mathbf{X}_I) - f(\mathbf{0})] \cdot \mathbb{E}[g(\mathbf{X}_J) - g(\mathbf{0})]$, which gives $\mathbb{E}[f(\mathbf{X}_I) g(\mathbf{X}_J)] \leq \mathbb{E}[f(\mathbf{X}_I)] \cdot \mathbb{E}[g(\mathbf{X}_J)]$, as required. ■

Corollary 2 *The random variables $\{X_u : u \in \widehat{E}\}$ are negatively associated.*

Proof of Lemma 8: Recall that Z is the capacity for each bucket in the binary ORAM tree, and R is the number blocks overflowing from the binary tree that need to be stored in the stash. The quantity that we wish to analyze is $\mathsf{u}^T(\mathsf{ORAM}^\infty[\mathbf{s}]) = \sum_{u \in \widehat{E}} X_u$.

We next perform a standard moment generating function argument. For $0 < t \leq \frac{1}{4}$, $\Pr[\mathsf{u}^T(\mathsf{ORAM}^\infty[\mathbf{s}]) \geq nZ + R] = \Pr[e^{t \sum_{u \in \widehat{E}} X_u} \geq e^{t(nZ+R)}]$, which by Markov's Inequality, is at most $E[e^{t \sum_{u \in \widehat{E}} X_u}] \cdot e^{-t(nZ+R)}$, which, by negative association (Corollary 2), is at most $\prod_{u \in \widehat{E}} \mathbb{E}[e^{tX_u}] \cdot e^{-t(nZ+R)}$. Putting $Z = 5$, and

$t = \ln \frac{5}{3} \leq \ln 2$, using Lemma 9, we conclude that the probability is at most $3 \cdot \frac{1}{4^n} \cdot (0.93312)^n \cdot 0.6^R$, as required. ■

Proof of Theorem 3. By applying Lemma 8 to inequality (1), we have the following:
$\Pr[\mathsf{st}(\mathsf{ORAM}^Z[\mathsf{s}]) > R] \leq \sum_{n \geq 1} 4^n \cdot 3 \cdot \frac{1}{4^n} \cdot (0.93312)^n \cdot 0.6^R \leq 42 \cdot 0.6^R$, as required.

B.2 Slight Improvement in Analysis Using Deterministic Eviction

Bottleneck for Measure Concentration. In Lemma 9, we see the bottleneck that puts a lower bound on the bucket size is caused by the moment generating function of X_u associated with an exit node $u \in E$ (whose parent is an internal node in the binary ORAM tree).

In order for the proof to work, we look for the smallest bucket size Z such that there exists some $t > 0$ such that for all $u \in \widehat{E}$, $4\mathbb{E}[e^{t(X_u - Z)}] < 1$. It can be checked that for a leaf $u \in F$, it is possible to set $Z = 4$ and $t = 1$ such that the above term is at most $\frac{1}{2}$; hence, we try to improve the eviction process when an exit node $u \in E$ is involved.

Intuitively, the moment generating function measures how much a random variable varies; hence, we could get a better measure concentration bound if we remove some randomness from the process.
Deterministic Eviction. Recall that in each round, one random eviction path from each of the left and the right branch is picked. Instead of picking the eviction paths randomly, the path at each branch can be picked deterministically in a way such that if a (non-root) exit node $u \in E$ has weight p_u, then the eviction path will intersect u once in exactly $\frac{1}{2p_u}$ rounds.

The modified process will induce a Markov process with non-negative state X_u, which represents the number of blocks residing in the (proper) ancestors of node u immediately after the step in which the eviction path intersects u. Specifically, the following occurs in each phase (which corresponds to $\frac{1}{2p_u}$ steps of ORAM operations).

- A random number M of items arrive, where in this case M follows the binomial distribution of $\frac{1}{2p_u}$ independent trials, each of which has success probability p_u.
- If the state is positive (possibly because items have just arrived), then exactly one item departs.

Denote $\widehat{X}_u$ as the stationary distribution of the Markov process. Then, the result from [28] implies that $\mathbb{E}[e^{t\widehat{X}_u}] = \frac{(1-\mathbb{E}[M])\cdot(e^t-1)}{e^t - \mathbb{E}[e^{tM}]} \leq \frac{1}{2} \cdot \frac{e^t-1}{e^t - \exp[\frac{1}{2}(e^t-1)]}$,
where we have used $\mathbb{E}[M] = \frac{1}{2}$, and $\mathbb{E}[e^{tM}] = (1 + p_u(e^t-1))^{\frac{1}{2p_u}} \leq \exp[\frac{1}{2}(e^t-1)]$, which is at most e^t for $t \in [0, 1.2]$.

It can be checked that setting $Z = 3$ and $t = 1$, we have $4\mathbb{E}[e^{t(\widehat{X}_u - Z)}] \leq \frac{1}{2}$. However, observe that $\widehat{X}_u$ represents the number of blocks above node u just after a step when the eviction path intersects u. Hence, just before eviction, the number of blocks is at most 1 larger. Therefore, using bucket size 4 is sufficient.

It can be checked that for a subtree T with n buckets and $Z = 4$,
$\Pr[\mathsf{u}^T(\text{ORAM}^\infty[\mathsf{s}])] > n \cdot Z + R] \leq 7 \cdot \frac{1}{4^n} \cdot (\frac{1}{2})^n \cdot e^{-R}$.

Hence, a similar calculation gives $\Pr[\mathsf{st}(\mathsf{ORAM}^Z[\mathsf{s}]) > R] \leq \sum_{n \geq 1} 4^n \cdot 7 \cdot \frac{1}{4^n} \cdot (\frac{1}{2})^n \cdot e^{-R} \leq 14 \cdot e^{-R}$, as stated in Theorem 4.

C Circuit Size of Circuit ORAM

It can be seen easily that the circuit size for $\mathsf{ReadAndRm}$ is $O(D \log N) \cdot \omega(1)$ and the circuit size to add a block to the stash is $O(D \log N) \cdot \omega(1)$. It remains to calculate the circuit size of eviction. Eviction can be done in a circuit of size $O(D \log N + \log^2 N) \cdot \omega(1)$ given the following fact mentioned in Wang *et al.* [47].

Fact 6 *Given Z blocks of a bucket denoted $\{\mathtt{idx}_i || \mathsf{label}_i || \mathtt{data}_i\}_{0 \leq i \leq Z}$ and the leaf label for the current eviction path, there is a circuit of size $O(Z \log N)$ that finds the deepest block w.r.t. to the eviction path.*

Let $\mathsf{B}_i := \{\mathtt{idx}_i || \mathsf{label}_i || \mathtt{data}_i\}$ denote the i-th block. Let $\mathtt{p}$ denote the leaf label of the eviction path. It is not hard to see that block B_i can be placed deepest along the path while maintaining the main invariant if and only if $\mathsf{label}_i \oplus \mathtt{p}$ has more leading zeros. However, counting number of leading zeros of an L-bit string requires a circuit of size $O(L \log L)$. Instead of counting the number of leading zeros and finding the maximum value in a bucket, we use an alternative method. For a bit string s, we denote s' to be the string constructed by setting all bits lower than the most significant one bit as one. The idea is based on the fact that $\mathit{leading\_zero}(s_1) > \mathit{leading\_zero}(s_2)$ if and only if $s'_1 < s'_2$. So, instead of counting number of leading zeros and find the maximum value, we can 1) for each label_i, compute label'_i by setting all bits lower than the most significant one bit as one; 2) find the block with minimum label'_i.

D More Details on the Goldreich-Ostrovsky Lower Bound

In this section, we elaborate on why the Goldreich-Ostrovsky lower bound works for even $O(1)$ failure probabilities. The argument is simple, but we write it down for completeness.

Based on the Goldreich-Ostrovsky lower bound proof, we define a p-long physical access sequence to be one that is compatible with at least p fraction of logical request sequences of length t. If $0 < p < 1$ is a constant, then a p-long physical sequence must be of length $q = \Omega(t \log N)$.

Definition 8 (Sufficiently long physical access sequence) *Let $0 < p < 1$ denote a constant. A physical access sequence is p-long, if $c^q > pN^t$, where c is an appropriate constant, q is the length of the physical access sequence, and t is the runtime of the non-oblivious RAM.*

When a physical access sequence is not p-long, we say that it is p-short.

Consider the following stochastic process: pick a random logical request sequence of length t, run the ORAM simulation. In this stochastic process, randomness is defined with respect to the choice of the logical sequence, as well as the ORAM's randomness.

We would like to show the following theorem:

Theorem 7 *Let $0 < p < 1$ denote a constant. For any ORAM with $(1 - p)/2$ failure probability, then, with probability $1/2$ in the above stochastic process, the physical access sequence must be p-long.*

Proof: We now prove by contradiction. Assume that with probability more than $\frac{1}{2}$, the physical access sequence is p-short.

We construct the following adversary and show that it can win the ORAM game with $(1-p)/2$ probability. The adversary picks two random logical access sequences of length t, and gives them to the challenger. The challenger picks a logical sequence at random, runs the ORAM simulation, and returns the physical access sequence to the adversary.

Conditioned on the physical access sequence being p-short, the probability that the physical access sequence is compatible with the other (i.e., not chosen by the challenger) logical sequence is bounded by p. Therefore, the adversary can win the ORAM security game with probability $(1 - p)/2$. ■