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

May 15, 2015

Abstract

We propose a new tree-based ORAM scheme called Circuit ORAM. Circuit ORAM makes both theoretical and practical contributions. From a theoretical perspective, Circuit ORAM shows that the well-known Goldreich-Ostrovsky logarithmic ORAM lower bound is tight under certain parameter ranges, for several performance metrics. Therefore, we are the first to give an answer to a theoretical challenge that remained open for the past twenty-seven years. Second, Circuit ORAM earns its name because it achieves (almost) optimal circuit size both in theory and in practice for realistic choices of block sizes. We demonstrate compelling practical performance and show that Circuit ORAM is an ideal candidate for secure multi-party computation applications.

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 [11–13, 36, 46], cloud outsourced storage [20, 48, 49, 56] and secure multi-party computation [14, 15, 24, 27, 32, 52].

1.1 Rethinking ORAM Metric for Secure Computation

When ORAM was previously considered for cloud outsourcing and secure processor applications, its *bandwidth cost* [20, 47, 49–51] was used as a primary metric of performance, since it is well-understood that bandwidth is the main performance bottleneck in these scenarios. As a result, many existing ORAM schemes focused on optimizing the bandwidth metric [20, 47, 49–51].

With the new tree-based ORAM framework, it became feasible to implement ORAM atop secure multi-party computation (MPC) [24, 27, 52]. It is well-understood that ORAM carries the promise of scaling MPC to big data [32, 52].As a result, the community is interested in optimized ORAM constructions for MPC [14, 24, 27, 35, 52]. One recent revelation [52] is that ORAMs optimized for the bandwidth metric are not necessarily the best for the MPC scenario. Instead, MPC demands a new metric of ORAM schemes, namely, the *circuit complexity*. In a standard ORAM setting, a client interacts with a server multiple rounds to make an ORAM access, and performs computation in between these accesses. In an MPC scenario, the ORAM client's computation will be expressed as

circuits that will be securely evaluated between the multiple parties. Therefore, an ORAM's circuit complexity is the total circuit size of the ORAM client algorithm over all rounds of interaction [52]. Furthermore, for several MPC protocols [17, 57] where XOR operations are essentially free [29], the number of AND gates is the primary performance metric.

1.2 The Quest for ORAMs with Optimal Circuit Complexity

Due to increasing interest of implementing ORAMs to scale up MPC, the hunt is on for an ORAM scheme with optimal circuit complexity. Our new ORAM scheme, Circuit ORAM, is the first to give a compelling solution to this question in both practical and theoretical senses.

Compelling practical performance. In comparison with known ORAM schemes, Circuit ORAM achieves 58.4X improvement in terms of number of non-free gates [29] over a straightforward implementation of Path ORAM [51], 31X improvement over the binary-tree ORAM [47], and 5.1X improvement over the recent SCORAM [52] – a heuristic ORAM scheme without theoretical performance bounds. These performance numbers are attained for a moderately large database of 4GB and with a 2^{-80} security failure probability. Our performance gains are asymptotic, so Circuit ORAM's improvement will be even greater for bigger data sizes.

Several earlier works on RAM-model MPC also investigated at what data sizes ORAM starts to outperform the trivial linear-scan ORAM, a metric referred to as the *breakeven point* with trivial ORAM. Previous implementations reported this breakeven point to be rather large. For example, Gordon *et al.* [24] who implemented the binary-tree ORAM [47] reported a breakeven point of 8MB with a block size of 512 bits and a security failure probability of roughly 2^{-14}. Under the same block size, we achieve an 8KB breakeven point (an 1000X improvement) at a much tighter failure probability of 2^{-80}!

Theoretical near-optimality. For block sizes of $D = \Omega(\log^2 N)$ bits or higher, Circuit ORAM achieves a circuit size of $O(D \log N)\omega(1)$ gates for a $\mathsf{negl}(N)$ failure probability[1]. This is almost optimal since the well-known logarithmic ORAM lower bound [18] is immediately applicable to the circuit size metric as well. We discuss situations when the logarithmic ORAM lower bound [18] is tight in the following section.

Table 1 compares the circuit size of Circuit ORAM and existing works, both in terms of asymptotics and concrete performance numbers.

1.3 On Tightness of the Goldreich-Ostrovsky ORAM Lower Bound

For theoretical discussions regarding the tightness of the Goldreich-Ostrovsky ORAM lower bound, *we consider several additional metrics besides circuit size*, such as number of accesses and bandwidth blowup. We elaborate on these various metrics in Section A.2.

In their seminal work, Goldreich and Ostrovsky [16, 18] showed a logarithmic lower bound for any ORAM construction. While not explicitly stated in their work, it is clear that this lower bound is very "powerful" in the sense that it holds *i)* for arbitrary block sizes, *ii)* for several relevant

[1] Throughout this paper, the notation $g(N) = O(f(N))\omega(1)$ denotes that for any $\alpha(N) = \omega(1)$, $g(N) = O(f(N)\alpha(N))$.

Scheme	Circuit Size (asymptotic)*	# AND gates (concrete)**
Hierarchical ORAMs		
GO96 [16, 18]	$O(D \log^3 N + C_{\text{PRF}} \log^2 N)$	≥476.1M
GM11 [20]	$O(D \log^2 N + C_{\text{PRF}} \log N)$	
KLO12 [31]	$O((D + C_{\text{PRF}}) \cdot \log^2 N / \log \log N)$	≥linear scan
LO13 [35] (Note: 2-server model)	$O((D + C_{\text{PRF}}) \cdot \log N)$	(63488M)
Tree-based ORAMs		
Binary-tree ORAM [47]	$O((D + \log^2 N) \log^2 N)\omega(1)$	30.1M
CLP13 [7] (naive circuit)	$O((D + \log^2 N) \log^3 N)\omega(1)$	37.9M
CLP13 [7] (w/ oblivious queue [38, 44, 58])	$O((D + \log^2 N) \log^2 N)\omega(1)$	37.9M
Path ORAM (naive circuit) [51]	$O((D + \log^2 N) \log^2 N)\omega(1)$	56.6M
Path ORAM (o-sort circuit) [52]	$O((D + \log^2 N) \log N \log \log N)\omega(1)$	41.4M
Circuit ORAM (This Paper)	$\mathbf{O((D + \log^2 N) \log N)\omega(1)}$	**0.97M**

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.
**: The concrete circuit size is calculated based on 4GB data with a 32-bit block size, with 2^{-80} security failure probability.

metrics including number of accesses, bandwidth blowup, and circuit size[2]; and *iii)* even when tolerating up to $O(1)$ statistical failure probability.

For mildly large block sizes of $\Omega(\log^2 N)$, Circuit ORAM achieves $O(D \log N + D \log \frac{1}{\delta})$ cost in terms of both *circuit size* and *bandwidth cost*, where δ is the failure probability. This means that for any $f(N) = \omega(1)$, there is an ORAM scheme that achieves $O(D \log N) f(N)$ circuit size or bandwidth cost, such that its statistical failure probability is bounded by some negligible function $\mathsf{negl}(N)$. In other words, for the circuit size or bandwidth cost metrics, **the Goldreich-Ostrovsky lower bound is asymptotically tight for $\Omega(\log^2 N)$ block size and negligible failure probabilities**. Equivalently, we rule out any $g(N)$ lower bound where $g(N) = \omega(\log N)$.

As we elaborate in Appendix B, we also show the tightness of the lower bound for the classical *runtime blowup* metric, but under a bigger block size of $\Omega(N^\epsilon)$ bits for an arbitrary constant $0 < \epsilon < 1$.

To summarize, one way to view our tightness result is the following: we give explicit and broad parameter ranges under which no asymptotically tighter lower bound can be proven. While this provides only a partial answer to the tightness of the Goldreich-Ostrovsky lower bound, we stress that for the past twenty-seven years, the tightness of the lower bound has not been demonstrated for any parameter ranges at all.

[2] For number of accesses and bandwidth blowup, the lower bound is applicable to $O(1)$ client storage.

1.4 Technical Highlights

Path ORAM has a complex eviction circuit. In the secure processor setting, Path ORAM [51] and its improved variants [12, 15, 46] offer the best performance in terms of bandwidth cost. Therefore, the first natural idea is to try out Path ORAM for the MPC setting too. Unfortunately, Path ORAM's eviction circuit is complex, and would result in $O(D \log^2 N)$ size with a naive implementation. Although Wang *et al.* noted that Path ORAM's eviction algorithm can be implemented with a circuit of size $O(D \log N \log N \log N)\omega(1)$ using oblivious sorting [19], they also show that oblivious sorting makes the practical performance even worse than the naive $O(D \log^2 N)$ implementation for typical parameter ranges.

One thing to note is that Path ORAM's eviction algorithm in some sense performs (oblivious) sorting on $\Theta(\log N)$ items (imagine that bucket size were 1). Therefore, to do better we need a fundamentally different idea.

Reducing eviction complexity. Our idea is to find an eviction circuit that is less complex than that of Path ORAM's, and yet preserves 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 [51] and its variants [12, 15], 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 leverage two additional metadata scans to precompute the foresight required, before beginning the real block scan. Our construction and proofs share common themes with Path ORAM [51] and the CLP ORAM [7]. However, our construction and proofs *differ in a substantial and non-trivial manner from either Path ORAM or CLP ORAM.*

1.5 Related Work

Hierarchical ORAMs. 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 [6, 16, 18, 20–23, 31, 34, 35, 41–43, 53–56]. Most of these hierarchical ORAM schemes are expensive in practice for MPC applications, not only due to their asymptotical poly-logarithmic cost (as opposed to logarithmic), but also crucially, because the ORAM client in these schemes must compute a PRF function – in an MPC application, this PRF would have to be securely evaluated using a multi-party protocol. Further, all schemes dependent on cuckoo hashing [20, 31, 35] (including the two-server ORAM by Lu and Ostrovsky [35]) require the smallest level to have size $\Omega(\log^7 N)$ to tightly bound the failure probability of cuckoo hashing. Technically, this means that these schemes basically reduce to the trivial ORAM (or the Goldreich-Ostrovsky ORAM [16, 18]) for $N < 2^{37}$.

Tree-based ORAM framework. The tree-based ORAM framework, initially proposed by Shi *et al.* [47], departs fundamentally from the hierarchical framework [18], and is a new paradigm for constructing a class of ORAM schemes. Several later works [7, 14, 51] improved Shi *et al.*'s initial construction [47] (commonly referred to as binary-tree ORAM). These schemes are conceptually simpler, statistically secure, and easy to implement in secure processors [11–13, 36, 46] or MPC applications [15, 24, 27, 32, 52]. 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 [5] show a super-logarithmic lower bound for oblivious branching programs. Although some works cited their lower-bound as being applicable to the ORAM setting [8, 20], 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 [5], 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.

Oblivious storage and server-side computation. The original ORAM model proposed by Goldreich and Ostrovsky assumes a passive server (or memory) that does not perform computation. However, several subsequent works leveraged server-side computation to improve performance [2, 9, 37, 45, 56] or reduce the number of roundtrips [55].

To distinguish this server-computation setting from standard ORAM, we refer to it as *oblivious storage* as suggested by several earlier works [3, 6]. Apon *et al.* [3] point out that the *Goldreich-Ostrovsky lower bound is not applicable to the oblivious storage setting for the bandwidth metric.* In fact, one can construct oblivious storage schemes with constant bandwidth blowup but with poly-logarithmic server work. [2].

Most of existing oblivious storage schemes (with server computation) are unsuitable for the MPC setting, because they focus on optimizing the client-server bandwidth while paying the price of higher, typically poly-logarithmic server work. In an MPC setting, all server-side work must be securely evaluated using an MPC protocol which immediately incurs poly-logarithmic circuit size, and is thus expensive.

Subsequent work. Circuit ORAM is now provided as the default ORAM implementation in the ObliVM secure computation framework by Liu *et al.* [33]. ObliVM is based on garbled circuits, and at the front-end provides expressive programming abstractions and language features for non-specialist programmers. Using Liu *et al.*'s ObliVM framework, in Appendix E we provide more detailed end-to-end performance of Circuit ORAM.

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

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

2 Preliminaries

Definitions. Our definition of Oblivious RAM (ORAM) is standard, and we therefore defer formal definitions to Appendix A.1. Below, we introduce the tree-based ORAM framework.

2.1 Tree-based ORAM Framework

Shi *et al.* proposed a new tree-based framework [47], which was adopted subsequently by several improved constructions [7, 14, 39, 51, 52]. We now briefly review the framework.

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 $L = \log N + 1$ 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 assume that leaves are numbered sequentially from 0 to $N-1$. We also denote $[a..b] := \{a, a+1, \dots, b\}$.

Data structure. The server organizes *blocks* into a binary tree of height $L = \log N + 1$; each node of the tree is a *bucket* containing Z blocks. Each block is of the form:

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

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 [47], 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.

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.

As mentioned in Appendix B, we will leverage the "big block, little block" trick first proposed by Stefanov *et al.* [51] to parametrization the recursion, such that the recursion does not introduce additional asymptotic cost in terms of circuit size.

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 1 and 2, 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 structure as in the binary-tree ORAM [47] or Path ORAM [51] – 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*, block* B_0 *is deeper than block* 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.*

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 ≥ 1 do:
3:     if path[i] has empty slot then
4:         (B, ℓ) := Deepest block in path[0..i − 1] that can legally reside in path[i].
               /* B := ⊥ if such a block does not exist.*/
5:     end if
6:     if B ≠ ⊥ then
7:         Move B from path[ℓ] to path[i].
8:         i := ℓ      // skip to level ℓ
9:     else
10:        i := i − 1
11:    end if
12: end while
```

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 [7] – but it will soon become apparent that we apply it in a fundamentally different manner.

3.2 Intuition

We would like to have an eviction algorithm that is easy to implement as a small circuit. Ideally it should make *a single scan* of the data blocks on the eviction path from the stash to leaf (and only a constant number of metadata scans), and still try to push blocks towards the leaf as much as possible.

During the one-pass scan of the data blocks, 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.

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

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$[i]$. **/*

Initialize deepest $:= (\perp, \perp, ..., \perp)$, src $:= \perp$, goal $:= -1$.
if stash not empty **then**
    src $:= 0$,
    goal $:=$ Deepest level that a block in path$[0]$ can legally reside on path.
end if
for $i = 1$ to L **do**:
    if goal $\geq i$ **then** deepest$[i]$ $:=$ src
    end if
    $\ell :=$ Deepest level that a block in path$[i]$ can legally reside on path.
    if $\ell >$ goal **then**
        goal $:= \ell$, src $:= i$
    end if
end for

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. If the bucket size is $O(1)$, then the bandwidth blowup is $O(\log N)$. The most technical part of the proof is to show that the stash size is still $O(\log N)$ with similar failure probability as Path ORAM.

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 path$[i]$, it will try to evict the deepest block B in path$[0..i-1]$ to this empty slot, provided that the block B can legally reside in path$[i]$. Suppose this deepest block B resides in path$[\ell]$ where $\ell < i$. After relocating the block B to path$[i]$, the algorithm now skips levels path$[\ell+1..i-1]$, and continues its reverse scan at level ℓ instead (Line 8 in Algorithm 1). In case no block in path$[0..i-1]$ can fill the empty slot in path$[i]$, the scan simply continues to level 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

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

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

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.

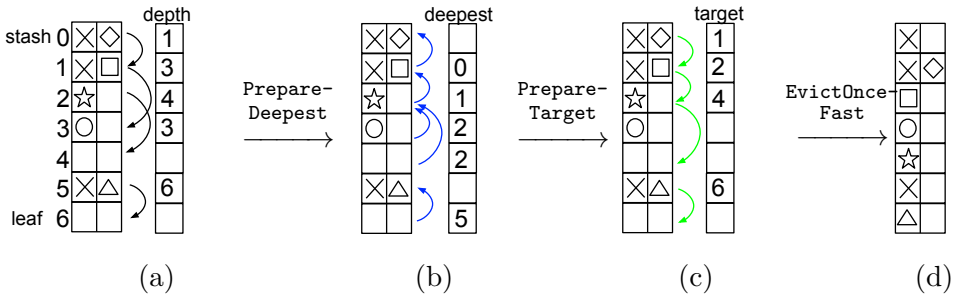


Figure 2: **An example of Circuit ORAM eviction.** Consider an eviction path with the stash at level 0, and the leaf at level 6. Bucket size and stash size are both 2. When there are two blocks in a level, $\times$ represents a block that is the *less deep*. The eviction will not care about the $\times$ blocks, but all other blocks are of potential interest, so we use distinct shapes to distinguish them.
At the beginning, `depth[i]` contains the deepest level of blocks in level i. The results of the two metadata scans are stored in the arrays `deepest` and `target` respectively. To aid understanding, we use arrows to visualize the arrays `depth`, `deepest`, and `target`. More detailed explanations of the arrows in each subfigure are provided below.
(a) $s \to t$: A black arrow from level s to level t means the following: a block in $\mathsf{path}[s]$ can legally reside in $\mathsf{path}[t]$; but no block in $\mathsf{path}[s]$ can legally reside in $\mathsf{path}[t+1..L]$. Here $s < t$.
(b) $s \to t$: A blue arrow from level s to level t means the following: the deepest block in $\mathsf{path}[0..s-1]$ that can legally reside in $\mathsf{path}[s]$ currently resides in $\mathsf{path}[t]$. Here $t < s$.
(c) $s \to t$: A green arrow from level s to level t means the following: during the real block scan, the client should pick up the deepest block in $\mathsf{path}[s]$, and drop it in $\mathsf{path}[t]$. Here $s < t$.
(d) Blocks are evicted according to `target` pointers (in green).

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.* [14] and several subsequent works [12, 15].

Recursion. So far, we have assumed that the client stores the entire position map. Based on a standard trick [47, 50, 51], 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.* [51]. Specifically, we group c number of `labels` in one block for an appropriate constant $c > 1$. In other words, the block size for position map levels is set to be $D' = O(\log N)$, resulting in $O(\log N)$ depth of recursion. 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 $O(D \log N + \log^3 N)$.

Security proof. The security proof is trivial. First, as in all tree-based ORAMs, every time a block is read or written, a random path is read, where the random choice has not been revealed to

Algorithm 4 `EvictOnceFast(path)`

Call the `PrepareDeepest` and `PrepareTarget` subroutines to pre-process arrays deepest and target.
hold := $\perp$, dest := $\perp$.
for $i = 0$ to L **do**
    towrite := $\perp$
    if (hold $\neq \perp$) and (i == dest) **then**
     /* *The block stored in* hold *will be placed in bucket* path$[i]$. */
        towrite := hold
        hold := $\perp$, dest := $\perp$.
    end if
    if target$[i] \neq \perp$ **then**
        hold := read and remove deepest block in path$[i]$
        dest := target$[i]$
    end if
    Place towrite into bucket path$[i]$ if towrite $\neq \perp$.
end for

Algorithm 5 `EvictRandom()`

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.
Call EvictOnceFast(path_0) and EvictOnceFast(path_1)

Algorithm 6 `EvictDeterministic()`

In timestep t:
Choose two paths, path_0 and path_1, corresponding to the leaves labeled with integers bitrev($2t \mod N$) and bitrev($(2t+1) \mod N$), respectively. In the above bitrev(i) denotes the integer obtained by reversing the bit order of i when expressed in binary.
Call EvictOnceFast(path_0) and EvictOnceFast(path_1)
Increase t by 1 for the next access.

the server before. This part of the proof is trivial, and the same as Shi *et al.* [47]. We now show that the eviction process is oblivious too. As we can see from Algorithms 2,3 and 4, eviction on a selected path always reads blocks or metadata (stored on the server) in a sequential manner, either from leaf to root or from root to leaf. Clearly this does not depend on the logical address being read or written. In fact, saying that our eviction algorithm (Algorithms 4) is oblivious is the same as saying that it can be implemented efficiently in circuit representation! Finally, no matter whether we use random-order eviction (Algorithm 5) or deterministic order eviction (Algorithm 6), the choice of the eviction path is also independent of the logical address sequence being read/written.

3.4 Theoretical Bounds

Although our scheme superficially borrows the "deepest" idea from the CLP ORAM, and borrows the "eviction on a path" idea from Path ORAM, *we stress that our construction is fundamentally*

different from either which necessitates novel proof techniques.

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 C 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 C for more details).

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.

Stash bounds. We prove stash bounds for both random-order and deterministic-order eviction. Below we give the formal theorem statements but defer their proofs to the appendices.

Theorem 1 (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.

The detailed proof of this theorem is deferred to Appendix C and D.1. .

Theorem 2 (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.

The detailed proof of this theorem is deferred to Appendix C and D.2. .

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 3 (Circuit size bound) *Circuit ORAM achieves $O((D+\log^2 N)(\log N+\log \frac{1}{\delta}))$ circuit size for a statistical failure probability of δ. Specifically, if the block size $D = \Omega(\log^2 N)$, then Circuit ORAM achieves $O(D(\log N + \log \frac{1}{\delta}))$ circuit size for a statistical failure probability of δ.*

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). In Theorems 1 and 2, we show that the stash size is $O(\log \frac{1}{\delta})$ blocks for a failure probability of δ. The rest immediately follows. ■

4 Proof Roadmap

We give a roadmap of how the probability statements concerning stash usage in Theorems 1 and 2 are proved. The full proofs are given in Appendices C and D. Although our proof borrows some high-level ideas from both Path ORAM [51] and CLP ORAM [7], we stress that *both our construction and proof are non-trivial and differ from Path ORAM and CLP ORAM in significant ways.*

Equivalence to post-processed ∞-ORAM. Similar to Path ORAM [51]'s analysis, we consider an imaginary construct known as ∞-ORAM, which is the same as Circuit ORAM, except that buckets have infinite capacity. ∞-ORAM is not a real-world efficient ORAM scheme, but a construct that facilitates analysis.

Suppose we are given the state of the ∞-ORAM at some moment, and we would like to infer from it the stash usage of the real ORAM. One natural way is to post-process the ∞-ORAM such that if a bucket contains more blocks than its capacity, the extra blocks are pushed back to its parent. This is performed repeatedly until all extra blocks are pushed from the root to the stash. As in the analysis of Path ORAM, the hope is that the stash usage of the post-processed ∞-ORAM is the same as that of the real ORAM. If this is true, then we can use the same approach [51] to analyze ∞-ORAM.

It is intuitive that the stash usage of the real ORAM should be at least that of the post-processed ∞-ORAM. Our ∞-ORAM shows how far blocks could be evicted towards the leaves even without any restrictions because of bucket capacity. The post-processing adds back the bucket capacity requirement. Hence, the post-processed ∞-ORAM gives some bound on how far the blocks could be evicted towards the leaves in the real ORAM.

However, it is not obvious that the real ORAM could evict blocks towards the leaves to the same extent as the post-processed ∞-ORAM. Indeed, if we do not perform partial eviction on the read path in a ReadAndRm operation (see Appendix C.1), then a hole (an available slot that would be filled in post-processing) could be created in the real ORAM. This could mean an extra block has to be stored in the stash of the real ORAM, thereby breaking the equivalence of stash usage between the real ORAM and the post-processed ∞-ORAM.

Fortunately, a partial eviction in a ReadAndRm operation is sufficient to fix this issue. However, Path ORAM's analysis, in our case it is highly non-trivial to show that indeed the post-processed ∞-ORAM is equivalent to the real Circuit ORAM. The key insight is to maintain the invariant (Fact 2 in Appendix C) that if some bucket in the real ORAM has an available slot, then during the post-processing of ∞-ORAM, no blocks can be pushed through this bucket towards the root. However, to formalize this argument requires an intricate induction proof that is presented in Appendix C. This technical proof can be skimmed, if the reader is convinced of the validity of the post-processed ∞-ORAM.

Probabilistic tools to analyze ∞-ORAM. Once it is established that the post-processed ∞-ORAM is equivalent to the real ORAM as in the analysis of Path ORAM [51], one can observe that the post-processed ∞-ORAM has stash usage of R blocks *iff* there exists a subtree T at the root with $n := n(T)$ buckets in ∞-ORAM such that the number of blocks residing in T is $nZ + R$, where Z is the bucket capacity. The goal is to show that at some fixed moment, for a fixed subtree T, the probability that T (in unprocessed ∞-ORAM) contains at least $nZ + R$ blocks is at most $\exp(-Cn - R)$, for some large enough constant $C > 0$. Since there are at most 4^n subtrees with n

nodes, a union bound over all possible subtrees T can establish the probability bound in Theorems 1 and 2.

The full proof is in Appendix D, and we outline the key ideas here. To analyze the usage of the subtree T, we consider two cases at the subtree's boundary, where blocks might possibly be evicted from T.

- Suppose bucket u in T is also a leaf in ∞-ORAM, and let X_u be the number of blocks in T whose label corresponds to u. Observe that these blocks cannot leave T. Since there are N distinct blocks, by a standard balls-into-bins argument, in the worst case, X_u is a sum of N independent $\{0,1\}$-random variables each having mean $\frac{1}{N}$.
- Suppose bucket u is not in T, but its parent bucket is in T. In this case, we say that u is an *exit* node, and let X_u be the number of blocks in T that can legally reside in u. Since in each ORAM access operation, a block is assigned a fresh random label and there are two eviction paths, we shall argue that X_u can be viewed as a Markov queue whose departure rate is twice that of its arrival rate. For random eviction path selection, X_u behaves like a a discrete-time M/M/1 queue, while for deterministic-order eviction variant, X_u behaves like a discrete-time M/D/1 queue [28]. Assuming that Circuit ORAM is initially empty, we can instead consider the stochastically dominating scenario when X_u is already in the stationary distribution, whose analysis does not depend on the number N of distinct blocks.

We shall see that in either of the above cases, the random variable X_u has constant expectation. Hence, the number of blocks in T, which is the sum of the X_u's, has expectation $\Theta(n)$. Observe that in the analysis of CLP ORAM [7], they consider how often each X_u reaches some threshold, whereas we directly consider the sum of the X_u's as in [51]. We next use a measure concentration argument to prove that the probability that the sum deviates from its mean is exponentially small.

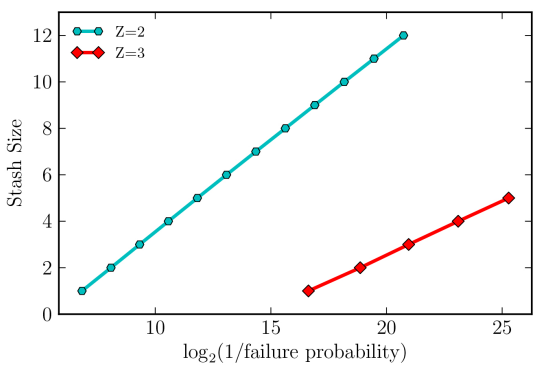
Observe that the X_u's are not independent. In fact, the X_u's are *negatively associated* [10], because if a block is assigned to one of the X_u's, then it cannot be assigned to another. Nevertheless, negative associativity is enough to prove measure concentration results via moment generating functions, which are standard tools used to derive results like Chernoff and Hoeffding bounds.

In Appendix D, we formally prove that the X_u's are negative associated using Lemma 10, and give upper bounds for the moment generating functions $t \mapsto \mathbb{E}[\exp(tX_u)]$ of the X_u's in Lemma 9. Finally, the probability calculations to achieve Theorems 1 and 2 are completed in the proof of Lemma 8.

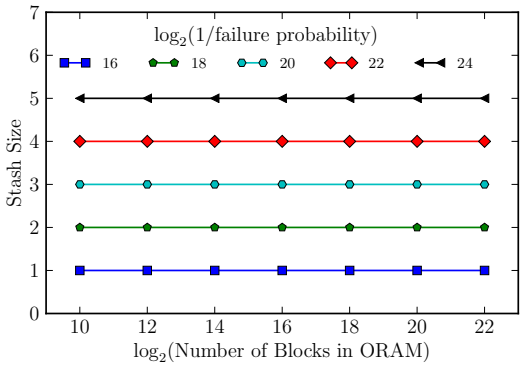
5 Evaluation

Stash size distribution. We simulate Circuit ORAM for a single long run, for about 2^{33} accesses, after 2^{25} accesses to warm up the ORAM to steady state. In our experiments, we use the following request sequence where we repeatedly cycle through all N logical addresses: $1, 2, \ldots, N, 1, 2, \ldots, N, \ldots$. Using the same argument as in Path ORAM [51], it is not hard to see that this is the worst-case access sequence. Instead of measuring multiple runs, we use a single, very long run, and measure the fraction of time that the stash has a certain size. This methodology is well-founded, since 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 [25]).

Figure 3a plots the stash size against the quantity $\log(\frac{1}{\delta})$ where δ is the failure probability. A point on the curve should be interpreted as: the stash exceeds R (value on y-axis) with probability



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



(b) **The stash size is independent of the ORAM capacity N.** $Z = 3$ is used.

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

Type Of ORAM	Circuit ORAM		Path ORAM(naive)		Path ORAM(o-sort)		SCORAM	
	Det.	Rand.	Det.	Rand.	Det.	Rand.	Det.	Rand.
Circuit size	$3.5M$	$6.6M$	$28.5M$	$170.1M$	$62.1M$	$124.1M$	$14.5M$	$17.9M$
Relative overhead	$1X$	$1.9X$	$8.2X$	$48.6X$	$17.7X$	$35.5X$	$4.14X$	$5.1X$
#AND gates	$0.97M$	$1.6M$	$9.5M$	$56.6M$	$20.7M$	$41.4M$	$4.9M$	$6.1M$
Relative overhead	$1X$	$1.6X$	$9.79X$	$58.4X$	$21.34X$	$42.7X$	$5.1X$	$6.3X$

Table 2: **Comparison of Circuit ORAM and variant of Path ORAM.** $N = 2^{30}, D = 32, \delta = 2^{-80}$. Path ORAM(o-sort) [52] uses 3 o-sorts. "Rand." stands for randomly chose eviction paths; "Det." stands for eviction with reverse-lexicographical-ordered paths, described in earlier works [12, 14, 15].

δ. The two curves correspond to a bucket size of 2 and 3 respectively. Clearly, the fact that both curves are a linear line suggests that the stash exceeds R with probability of 2^{-cR}, where the constant c in the exponent is different for the two curves. Further, Figure 3b shows that the stash size is independent of the ORAM's capacity N. Besides a bucket size of 2 and 3, we also tried a bucket size of 4 – in this case, we never observed the stash growing beyond 5 for the first 2^{33} accesses.

Circuit size. In Table 2, we compare the circuit sizes of Circuit ORAM and other state-of-the-art ORAM schemes. Results in this table are obtained for a 4GB dataset with the following concrete parameters: $N = 2^{30}$, $D = 32$ bits, and security failure $\delta = 2^{-80}$. For Path ORAM [51], we consider a naive implementation, and an asymptotically more efficient implementation relying on 3 oblivious sorts [52]. 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* [14]). The table shows that Circuit ORAM results in 8.2x to 48.6x smaller circuit size than Path ORAM, and is 4.1x to 5.1x better than SCORAM [52]. Circuit ORAM's speedup will become even bigger when the total data size N is greater.

	Circuit ORAM $\delta = 2^{-80}$						GKKKMRV12 [24] $\delta \approx 2^{-14}$	KS12 [27] $\delta = 2^{-20}$
Block size (bits)	32	**40**	128	**512**	2048	8192	512	40
Breakeven point (entries)	256	**256**	128	**128**	64	64	131072	≈2000
Breakeven point (total data size)	1KB	**1.3KB**	2KB	**8KB**	16KB	65KB	8MB	9.77KB

Table 3: **Breakeven point for different ORAMs.** Circuit ORAM numbers correspond to a security parameter of 80, whereas GKKKMRV12 and KS12 adopt a security parameter of roughly 14 and 20 respectively.

Breakeven point with trivial ORAM. Depending on the block size, the breakeven point between Circuit ORAM and trivial ORAM varies – Table 3. We also compare our breakeven point with Gordon *et al.* [24], and Keller and Scholl [27], and show that we achieve dramatic improvement at much higher security parameters. Gordon *et al.* implemented the binary-tree ORAM and reported a breakeven point of 8MB with a block size $D = 512$ bits, and 2^{-14} failure probability. Our breakeven point is 8KB with a block size $D = 512$ bits, and at 2^{-80} failure probability. Keller and Scholl [27] implement an optimized Path ORAM algorithm, and report a breakeven point of 9.77KB with a block size $D = 40$ bits, and a 2^{-20} failure probability. We achieve 1.3KB breakeven point at $D = 40$ bits, and with 2^{-80} failure probability.

Implementation over garbled circuits. Since our work, Circuit ORAM has now been implemented and provided as the default ORAM implementation in the ObliVM secure computation framework [1,33]. In Appendix E, we report Circuit ORAM's end-to-end performance over garbled circuits based on numbers gathered from the ObliVM framework [1,33].

References

[1] http://www.oblivm.com.

[2] Anonymous. Onion ORAM: A constant bandwidth oram without FHE, 2015.

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

[4] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 535–548, New York, NY, USA, 2013. ACM.

[5] 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.

[6] 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.

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

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

[9] J. Dautrich, E. Stefanov, and E. Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, San Diego, CA, Aug. 2014. USENIX Association.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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/.

[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. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC '14.

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

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] M. Keller and P. Scholl. Efficient, oblivious data structures for mpc. In P. Sarkar and T. Iwata, editors, *Advances in Cryptology ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages 506–525. Springer Berlin Heidelberg, 2014.

[28] 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*, 1953.

[29] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *International Colloquium on Automata, Languages and Programming*, 2008.

[30] 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.

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

[32] 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.

[33] C. Liu, X. S. Wang, E. Shi, Y. Huang, and K. Nayak. ObliVM: A Generic, Customizable, and Reusable Secure Computation Architecture. Manuscript, 2014.

[34] 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.

[35] S. Lu and R. Ostrovsky. Distributed oblivious ram for secure two-party computation. In *Proceedings of the 10th Theory of Cryptography Conference on Theory of Cryptography*, TCC'13, pages 377–396, Berlin, Heidelberg, 2013. Springer-Verlag.

[36] 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.

[37] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining oram and pir. 2014.

[38] 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.

[39] T. Moataz, T. Mayberry, E.-O. Blass, and A. H. Chan. Resizable tree-based oblivious ram. Financial Crypto, 2015.

[40] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 129–139, New York, NY, USA, 1999. ACM.

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

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

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

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

[45] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. Cryptology ePrint Archive, Report 2014/997, 2014. http://eprint.iacr.org/.

[46] 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.

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

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

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

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

[51] 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.

[52] 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.

[53] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

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

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

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

[57] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *IEEE symposium on Foundations of Computer Science (FOCS)*, 1982.

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

[59] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, 2015.

A ORAM Definitions and Metrics

A.1 Oblivious RAM Definitions

Notation. We use the notation $\{\mathsf{raddr}\}$ and $\{\mathsf{waddr}\}$ to refer to a set of *physical* addresses for reading and writing. We use $\{\mathsf{fetched}\}$ to denote a set of most recently fetched memory reads. Each *logical* operation $\mathsf{op} := (\mathtt{read}, \mathtt{idx})$ or $\mathsf{op} := (\mathtt{write}, \mathtt{idx}, \mathtt{data})$ either reads logical address $\mathtt{idx}$ or writes logical address $\mathtt{idx}$. Here we use the terminology memory and server interchangeably.

Oblivious RAM. An ORAM scheme can be formulated as a PPT algorithm denoted Next, commonly referred to as the ORAM client algorithm:

$$(\mathsf{out}, \{\mathsf{raddr}\}, \{\mathsf{waddr}\}, \{\mathtt{data}\}, st) \leftarrow \mathsf{Next}(\mathsf{op}, st, \{\mathsf{fetched}\})$$

Specifically, Next is a probablistic stateful algorithm (whose state is denoted st) that computes the physical memory operations for each round of interaction.

Given a sequence of memory access operations $\{\mathsf{op}_i\}_{i\in[M]}$, where each $\mathsf{op}_i := (\mathtt{read}, \mathtt{idx})$ or $\mathsf{op}_i := (\mathtt{write}, \mathtt{idx}, \mathtt{data})$ we can define the following execution:

oram-exec$((\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_M))$

Initialize $\mathsf{mem} := \{0, 0, \ldots, 0\}$; $\mathsf{res} := \emptyset$.
for $i = 1, 2, \ldots$ **do**
  $\{\mathsf{fetched}\} := \perp$
  repeat
    $(\mathsf{out}, \{\mathsf{raddr}\}, \{\mathsf{waddr}\}, \{\mathtt{data}\}, st) \leftarrow \mathsf{Next}(\mathsf{op}, st, \{\mathsf{fetched}\})$.
    $\{\mathsf{fetched}\} := \mathsf{mem}[\{\mathsf{raddr}\}]$; $\mathsf{mem}[\{\mathsf{waddr}\}] := \{\mathtt{data}\}$;
  until $\mathsf{out} \neq \perp$
  $\mathsf{res} := \mathsf{res}||\mathsf{out}$
end for

Correctness. An ORAM scheme is said to be correct if, for any input sequence $(\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_M)$, with probability 1, the result array res from **oram-exec** agrees with the result res from a true execution on the same input sequence.

true-exec$((\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_M))$

1: Initialize $\mathsf{mem} := \{0, 0, \ldots, 0\}$; $\mathsf{res} := \emptyset$.
2: **for** $i = 1, 2, \ldots$ **do**
3: **if** op_i is a `read` **then**
4: $\mathsf{res} := \mathsf{res} || \mathsf{mem}[\mathtt{idx}_i]$
5: **else**
6: $\mathsf{res} := \mathsf{res} || \mathtt{data}_i$; $\mathsf{mem}[\mathtt{idx}_i] := \mathtt{data}_i$
7: **end if**
8: **end for**

Security. Let $\mathsf{addr}(\textbf{oram-exec}(\mathsf{op}_1, \ldots, \mathsf{op}_M))$ denote the temporally ordered sequence of $\{\mathsf{raddr}\}$ and $\{\mathsf{waddr}\}$ output by the Next algorithm during the entire execution $\textbf{oram-exec}(\mathsf{op}_1, \ldots, \mathsf{op}_M)$. An ORAM scheme is said to be secure for any $M \in \mathbb{N}$, for any two sequences $(\mathsf{op}_1, \ldots, \mathsf{op}_M)$ and $(\mathsf{op}'_1, \ldots, \mathsf{op}'_M)$,

$$\mathsf{addr}(\textbf{oram-exec}(\mathsf{op}_1, \ldots, \mathsf{op}_M)) \equiv \mathsf{addr}(\textbf{oram-exec}(\mathsf{op}'_1, \ldots, \mathsf{op}'_M))$$

In particular, $\equiv$ denotes either statistical or computational indistinguishability. We refer to them as statistical or computational security respectively.

A.2 ORAM Metrics

We elaborate on the commonly used metrics for ORAM schemes. Since ORAM has been applied to several application settings such as storage outsourcing, secure processor, and MPC, several metrics have been adopted by the community. We stress that these metrics are related but distinct, and we elaborate on their definitions and relations below.

Bandwidth cost and bandwidth blowup. An ORAM's *bandwidth cost* refers to the average number of bits transferred for accessing each block of D bits. 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.

Number of accesses. The "number of accesses" metric characterizes how many times the ORAM client must access physical memory on average to satisfy each ORAM request. Two blocks at different addresses count as two distinct accesses even if they are accessed in the same roundtrip. This was the original metric considered by Goldreich and Ostrovsky in their original ORAM work (where they equivalently call it the runtime blowup comparing the Oblivious RAM simulation and the original non-oblivious RAM).

We note that the "number of accesses metric" is in fact the same as bandwidth blowup if block sizes are *uniform*. However, these metrics do not necessarily agree when blocks have *non-uniform* sizes, e.g., in recent tree-based ORAM schemes [47, 51], a "big data block, little metadata block" trick is commonly used to achieve better bandwidth costs.

Circuit size. The circuit size metric for ORAMs was first raised by Wang *et al.* [52], and is defined as the total circuit size of the ORAM client algorithm Next over all execution rounds during each ORAM request. Below are some simple but useful observations regarding the new circuit size metric [52], and relate the circuit size to the traditional bandwidth metric:

- 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 size, since the client work must be factored into the circuit size as well. Based on the above observation, the circuit size metric is a more "stringent" metric than the traditional bandwidth metric, since circuit size captures not only bandwidth but also client work (and server work too for oblivious storage schemes that require server computation).

B Interpreting Circuit ORAM under Other Metrics

Circuit ORAM also outperforms previous schemes in terms of bandwidth costs and number of accesses under wide parameter ranges. To obtain these costs, we first discuss ways to parametrize the recursion to get different costs under different block size assumptions.

Parametrizing the recursion. Recursion can be done using either *uniform* or *non-uniform* block sizes.

- *Non-uniform block size.* Of particular interest is when we set the block size of the recursive position map levels to be $D' = O(\log N)$, a standard "big data block, little metadata block" trick initially adopted by Path ORAM [51]. In this case, the recursion has $O(\log N)$ depth.

  This parametrization trick is good for optimizing circuit size and bandwidth cost – see Table 1 (Section 1) and Table 4 (this section). Specifically, in these tables, observe that the costs for tree-based ORAMs have two terms, where one term corresponds to the data level, and the other term corresponds to all metadata levels of the recursion.
- *Uniform block size.* In this case, we use the same block size for all recursion levels. A special case of interest later is when the block size is $D = N^\epsilon$ for some constant $0 < \epsilon < 1$ – in this case, the depth of the recursion is $O(1)$.

Circuit ORAM's bandwidth cost and number of accesses. We now state the cost of Circuit ORAM for the bandwidth and number of accesses metrics.

Theorem 4 (Circuit ORAM's bandwidth cost and number of accesses) *Circuit ORAM achieves the following bandwidth cost and number of accesses for a* $\mathsf{negl}(N)$ *statistical failure probability.*

- *Suppose the position map levels adopt a block size* $D' = O(\log N)$*, then Circuit ORAM achieves* $O(D \log N + \log^3 N)\omega(1)$ *bandwidth cost and* $O(\log^2 N)$ *number of accesses. In particular, for a* $D = \Omega(\log^2 N)$ *block size, the bandwidth cost is* $O(D \log N)\omega(1)$*.*

Scheme	Amortized Bandwidth Cost	Client Storage
	Hierarchical ORAMs	
GO96 [18]	$O(D \log^3 N)$	$O(D)$
GM11 [20]	$O(D \log^2 N)$	$O(D)$
KLO12 [31]	$O(D \log^2 N / \log \log N)$	$O(D)$
LO13 [35] (Note: 2-server model)	$O(D \log N)$	$O(D)$
	Tree-based ORAMs	
Binary-tree ORAM [47]	$O(D \log^2 N + \log^4 N) \cdot \omega(1)$	$O(D)$
CLP13 [7]	$O\left(\left(D \log N + \log^3 N\right) \log \log N\right)$	$O(D \log^2 N) \cdot \omega(1)$
Path ORAM(naive circuit) [51]	$O(D \log N + \log^3 N)$	$O(D \log N) \cdot \omega(1)$
Path ORAM(o-sort circuit) [52]	$O\left(\left(D \log N + \log^3 N\right) \log \log N\right) \cdot \omega(1)$	$O(D)$
Circuit ORAM (This paper)	$\mathbf{O(D \log N + \log^3 N) \cdot \omega(1)}$	$\mathbf{O(D)}$

Table 4: **Comparison of various ORAMs in terms of bandwidth cost.** The bounds are expressed for $\mathsf{negl}(N)$ 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 $D = \Omega(\log^2 N)$.

- *Suppose all levels adopt a uniform block size of $D = \chi \log N$, then Circuit ORAM achieves $O(D \log N \log_\chi N)\omega(1)$ bandwidth cost, and $O(\log N \log_\chi N)\omega(1)$ number of accesses. Of particular interest is when $D = N^\epsilon$ for some constant $0 < \epsilon < 1$. In this case, Circuit ORAM achieves $O(D \log N)\omega(1)$ bandwidth cost, and $O(\log N)\omega(1)$ number of accesses.*

Proof: The proof is immediate given the stash bound analysis (Theorems 1 and 2), and the stated methods for parametrizing the recursion. ■

Table 4 depicts the bandwidth cost of Circuit ORAM in comparison with existing works. For a realistic block size of $D = \Omega(\log^2 N)$, Circuit ORAM asymptotically outperforms all known schemes with $O(1)$ blocks of client-side storage.

Tightness of the Goldreich-Ostrovsky lower-bound for various metrics. Further, Theorem 3 (Section 3.4) and Theorem 4 also suggest that the Goldreich-Ostrovsky lower bound is tight under the following conditions and for a $\mathsf{negl}(N)$ statistical failure probability:

- For the circuit size and bandwidth metrics when the block size $D = \Omega(\log^2 N)$.
- For the "number of accesses" metric when the block size $D = N^\epsilon$ for a constant $0 < \epsilon < 1$.

C Analyzing Stash Size via Infinity ORAM

C.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.

C.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 $\mathsf{path}_R[\ell' + 1 \ldots \ell]$ 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.

Algorithm 7 `PartialEvictSlow(path, $\ell^*$)`

*/*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 := ℓ*      /* start from level ℓ* */
2: while i ≥ 1 do:
3:     if path[i] has empty slot then
4:         (B, ℓ) := deepest block and its level in path[0 ... i−1] that can reside in path[i] respecting
                     the path invariant.      /* B := ⊥ if such a block does not exist.*/
5:     end if
6:     if B ≠ ⊥ then
7:         Move B from path[ℓ] to path[i].
8:         i := ℓ
9:     else
10:        i := i − 1
11:    end if
12: end while
```

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.

C.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*[3], and there exists a block in $\mathsf{path}[0 \ldots \ell - 1]$ that can legally reside in $\mathsf{path}[\ell]$ or its descendant, then a block will move from $\mathsf{path}[0 \ldots \ell - 1]$ into $\mathsf{path}[\ell \ldots L]$.

  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 $\mathsf{path}[0 \ldots \ell - 1]$ that can legally reside in $\mathsf{path}[\ell \ldots L]$. Then, the block in

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

$\mathsf{path}[0 \ldots \ell - 1]$ 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 $\mathsf{path}[\ell \ldots L]$.

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

- **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.

C.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 $0 \leq a \leq b \leq L$, 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 $\mathsf{path}_R^k[a..b]$ is a sub-episode after the k-th step. Then, every block in $\mathsf{path}_\infty^k[a..b]$ must be in $\mathsf{path}_R^k[a..b]$ as well.*

Proof: We consider cases according to how the sub-episode $\mathsf{path}_R^k[a..b]$ is formed.
Case 1: $\mathsf{path}_R^{k-1}[a..b]$ 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): $\mathsf{path}_R^k[a..b]$ is a sub-episode resulting from splitting an episode $\mathsf{path}_R^{k-1}[\widehat{a}..b]$, for some $\widehat{a} < a$; during the k-th step, one block is removed from the full bucket $\mathsf{path}_R^{k-1}[a]$ to create a hole in $\mathsf{path}_R^k[a]$, 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 $\mathsf{path}_\infty^k[a..b]$ that is not in $\mathsf{path}_R^k[a..b]$.

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

Case 2(b): $\mathsf{path}_R^k[a..b]$ is a sub-episode resulting from splitting an episode $\mathsf{path}_R^{k-1}[a..\widehat{b}]$, for some $\widehat{b} > b$; during the k-th step, one block is removed from the full bucket $\mathsf{path}_R^{k-1}[b+1]$ to create a hole in $\mathsf{path}_R^k[b+1]$, 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 $\mathsf{path}^k_R[a..b]$. ■

D Analyzing Stash Usage of Infinity ORAM

In Section C, 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 [51], we use ORAM^Z to denote a real ORAM with bucket size Z, and ORAM^∞ to denote ∞-ORAM. Given an access sequence $\mathbf{s}$, the configurations of the ORAMs are denoted by $\text{ORAM}^Z[\mathbf{s}]$ and $\text{ORAM}^\infty[\mathbf{s}]$. The stash usage of the real ORAM is denoted as $\mathsf{st}(\text{ORAM}^Z[\mathbf{s}])$, and the stash usage of a post-processed ∞-ORAM to an ORAM with bucket size Z is denoted as $\mathsf{st}^Z(\text{ORAM}^\infty[\mathbf{s}])$. Then, the strong equivalence lemma (Lemma 1) states that $\mathsf{st}(\text{ORAM}^Z[\mathbf{s}]) = \mathsf{st}^Z(\text{ORAM}^\infty[\mathbf{s}])$.

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 $\mathsf{u}^T(\text{ORAM}^\infty[\mathbf{s}])$. An observation made in [51] is that for any $R \geq 0$, $\mathsf{st}^Z(\text{ORAM}^\infty[\mathbf{s}]) > R$ *iff* there exists a subtree T such that $\mathsf{u}^T(\text{ORAM}^\infty[\mathbf{s}]) > n(T) \cdot Z + R$, where $n(T)$ is the number of buckets in subtree T.

The following equations summarize the above discussion.

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

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 $\leq 4^n$,

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

Fixing some subtree T with n nodes (buckets), we next give a uniform upper bound for $\Pr[\mathsf{u}^T(\text{ORAM}^\infty[\mathbf{s}]) > nZ + R]$ in terms of n, Z and R. Although we make measure concentration argument similar to [51], the underlying random processes are actually very different. For instance, the blocks in [51] are evicted independently from one another, while the blocks in this case are dependent upon one another.

D.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

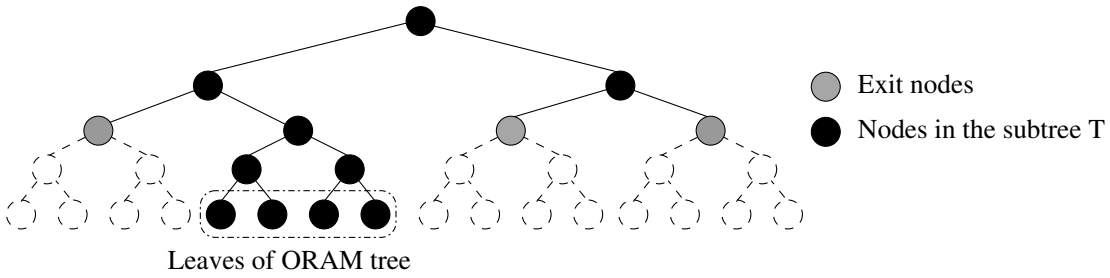


Figure 4: A subtree containing some leaves of the original ORAM binary tree, augmented with the exit nodes. This diagram is also used in the paper [51].

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.

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 [26], 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, we have $\prod_{u \in \widehat{E}} \mathbb{E}[e^{tX_u}] \leq (\frac{1}{2-e^t})^{n+1}$, 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 [10, 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 [10]) *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 [10, 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 [10, Lemma 8]. As mentioned in [10], 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 1 *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 1), 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 1. By applying Lemma 8 to inequality (1), we have the following:
$\Pr[\mathsf{st}(\mathsf{ORAM}^Z[\mathbf{s}]) > R] \leq \sum_{n \geq 1} 4^n \cdot 3 \cdot \frac{1}{4^n} \cdot (0.93312)^n \cdot 0.6^R \leq 42 \cdot 0.6^R$, as required.

D.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 [30] 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 $Z = 4$,

$\Pr[\mathsf{u}^T(\text{ORAM}^\infty[\mathbf{s}])] > n \cdot Z + R] \leq 7 \cdot \frac{1}{4^n} \cdot (\frac{1}{2})^n \cdot e^{-R}$.

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

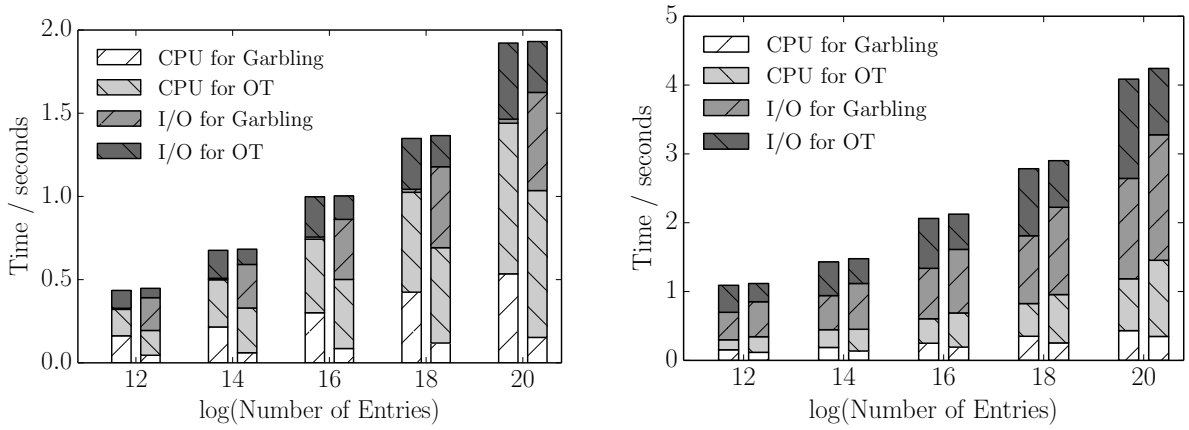
E Implementation over Garbled Circuits

As mentioned earlier, Circuit ORAM has now been implemented and provided as the default ORAM implementation in the ObliVM secure computation framework [1,33]. Liu *et al.* [1,33] report some Circuit ORAM performance numbers in the ObliVM framework. In this appendix, we report more detailed performance numbers of Circuit ORAM atop the ObliVM framework.

Running time. We tested the performance of Circuit ORAM under different network bandwidth configurations. Figure 5a and Figure 5b report the wallclock running time obtained, and the performance breakdown. To maximize performance, we apply different optimizations for these different settings. When the bandwidth is 1Gbps (Figure 5a), we use Garbled Row Reduction [40] and Free XOR [29]. When the bandwidth is 20Mbps, we use the halfgate optimization [59] instead.

Interpreting the performance numbers. First, the present implementation of the ObliVM framework does not exploit AES-NI instructions to speed up garbling. We expect a noticeable speedup for the computation time when hardware AES is employed. Second, the present ObliVM uses a Java-based implementation. Therefore, the timing measurements are subject to the artifacts of Java.

In Figure 5a, when the bandwidth is 1Gbps, we observe that the garbler's garbling I/O is negligible. The evaluator's garbling I/O is higher – and most of this stems from I/O synchronization cost since in this case the bandwidth is ample. When the bandwidth is 20Mbps (Figure 5b) the garbler spends noticeable time on transmitting garbled circuits to the evaluator (I/O for Garbling). Since garbling is never blocked waiting for the evaluator, most of this cost is network transmission cost. For the OT cost, presently ObliVM does not employ the known cache optimizations [4] (which are difficult to realize in a Java-based implementation). We therefore expect significant savings in performance when ObliVM transitions to a C-based implementation adopting state-of-the-art cache optimizations for OT, and hardware AES-NI features.



(a) **Bandwidth=1Gbps.** Garbled Row Reduction [40] and Free XOR [29] are used.

(b) **Bandwidth=20Mbps.** Halfgate [59] is used.

Figure 5: **Runtime breakdown for Circuit ORAM under different bandwidth configurations.** In each pair of bars, the *left* is for the *garbler* while the *right* is for the *evaluator*. Performance numbers are measured based on the ObliVM framework provided by Liu *et al.* [33]. The plots correspond to a data size $D = 32$ bits and $\delta = 2^{-80}$ failure probability.

F Supplemental Details

F.1 Circuit Size of Circuit ORAM: A More Detailed Analysis

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.* [52].

Fact 6 *Given Z blocks of a bucket denoted $\{\mathtt{idx}_i || \mathsf{label}_i || \mathtt{data}_i\}_{0 \le i \le Z}$ 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.

F.2 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 (and therefore it works for negligible or inverse-poly failure probabilities too). 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 5 *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 $(1-p)/2$. ■