SeqORAM: A Locality-Preserving Write-Only Oblivious RAM

Anrin Chakraborti and Radu Sion
Stony Brook University

July 6, 2017

Abstract

Oblivious RAM technology has advanced rapidly. Under certain client-side storage assumptions, tree-based ORAM designs [19] have achieved established lower bounds even for online adversaries that can monitor everything a client does.

Write-only ORAMs target a weaker but often more practical multi-snapshot adversary that can monitor only user writes. This adversary is typical in plausible deniability and censorship-resistant systems [5, 14].

But, although existing write-only ORAMs achieve significantly better theoretical performance than traditional ORAMs, these gains do not materialize when deployed on real systems. This happens primarily due to the random data placement strategies used to break correlations between logical and physical namespaces – thus ensuring write “unlinkability”, a required property. Random access performs poorly on both rotational disks and SSDs, often increasing wear significantly, and interfering with wear-leveling mechanisms.

In this work, we show how to achieve write “unlinkability” without random data access. SeqORAM is a new locality-preserving write-only ORAM. Data blocks close to each other in the logical domain land also in close proximity on disk. A full-fledged Linux kernel-level implementation of SeqORAM is 100x faster than the state-of-the-art for standard workloads.

1 Introduction

Dramatic advances in storage technology has resulted in users storing personal (sensitive) information on portable devices (mobiles, laptops etc).

Unfortunately, the stored information becomes vulnerable to unauthorized disclosures through attacks on the storage medium. “Drive snooping” – either through a potentially untrusted software installed on the system [1] or as firmware for the disk [3] – constantly monitors disk reads/writes and creates user profiles on the basis of the information being accessed more frequently.
Further, data privacy becomes significantly more important in the fight against increasing censorship and intrusion by unfriendly powerful nation state adversaries. There are numerous examples [14] of human rights activists being coerced to provide access and encryption keys to oppressive regimes.

To ensure confidentiality, data can be encrypted in between accesses and when written to disk. This however is not enough [11] because the sequence of locations read and written leaks information regarding the user’s access pattern and ultimately the contents of the stored data. This is especially important in providing plausible deniability mechanisms [5, 6].

To mitigate this, oblivious RAMs (ORAM) are designed to allow a client to hide data access patterns from an adversary monitoring a storage device or unsecured RAM. Informally, the ORAM adversarial model ensures computational indistinguishability between multiple equal-length query/access sequences. ORAMs also guarantee indistinguishability between reads and writes.

Although ORAMs provide an elegant solution for access privacy, they incur very significant overheads, often prohibitive in practical scenarios for securing local storage devices. Instead, a more common deployment scenario for ORAMs is to secure network storage where high network latency “masks” the access overhead inherent to ORAM designs. Thus, previous ORAM literature has largely focused on minimizing network bandwidth requirements and reducing the number of round trips to the server required in order to complete an access. In this regard, tree-based ORAM designs (such as PathORAM [19]) have achieved a known lower bound on bandwidth under assuming significant client side storage. Unfortunately, even in this case, ORAMs are impractical for deployment for most applications.

Further, ORAMs protect against an online adversary that can monitor all accesses. In practice this is often too strong when used for storage devices. Instead, a multi-snapshot adversary [5, 6, 14] that can access and observe the device after every write operation is of more interest.

The adversary can save and compare snapshots of the device, which may include device state information and filesystem metadata. The typical example involves customs officers at border crossings scanning a device under coercion, or malware monitoring disk writes/updates.

To overcome this, Li et al. [13] introduced write-only ORAMs that only protect privacy of write operations. Subsequent work by Blass et al. [5] has shown that write-only ORAMs can achieve significantly better performance when compared to ORAMs with both read and write privacy guarantees.

[5] also shows how to achieve plausible deniability of stored data using write-only ORAMs. Plausible deniability allows users to plausibly deny the existence of certain data stored on their device, against an adversary that may access and observe the device once (“single-snapshot”) or at multiple points in time (“multi-snapshot”).

Unfortunately, existing write-only ORAMs are still orders of magnitude slower than raw disks. For example, the state-of-the-art write-only ORAM, HIVE-ORAM [5] is almost four orders of magnitude slower for both reads and writes when compared to a SATA HDD and two orders of magnitude slower.
than SSDs.

The main cause of this performance penalty is an inherent random mapping while placing of data from logical to physical domains. More specifically, existing mechanisms such as HIVE-ORAM place logical data blocks in random physical locations to break correlations – thus ensuring “unlinkability” of write operations, an important required property of ORAMs. However, it is well known that random accesses can severely affect performance on both HDDs and SSDs. Further, this random placement prohibits any performance gains derived from modern filesystems optimized for sequential accesses.

In this paper, we introduce SeqORAM, a write-only ORAM that preserves locality of accesses and is orders of magnitude faster than the existing state-of-the-art.

SeqORAM is based on two key insights. First, instead of writing data to random locations in order to break correlations between logical and physical addresses, “unlinkability” can also be achieved by writing data in a canonical form, i.e., sequentially at increasing physical block addresses independent of their logical block addresses (similar to log-structured filesystems).

Second, to also speed up reads, data in blocks close together in the logical block ID realm are placed as close as possible on disk, thus significantly increasing locality and overall throughput when paired with modern locality-optimized filesystems.

SeqORAM is implemented as a Linux kernel device mapper. When compared with the state-of-the-art write-only ORAM [5], SeqORAM read and write throughputs are orders of magnitude faster for sequential accesses and standard workloads.

2 Related

ORAMs have been well-researched since the seminal work by Goldreich and Ostrovsky [8]. ORAM constructions can be broadly classified into two categories: pyramid-based and tree-based.

In ORAMs, data is structured as a set of \( N \) semantically-secure encrypted blocks (with a secret key held by the client). Two operations are supported: \( \text{read}(id) \) and \( \text{write}(id, value) \), where \( id \) is a logical identifier for the block being accessed.

**Pyramid based ORAM.** Pyramid ORAMs organize data into levels with each level exponentially larger than the other. The first pyramid based construction was provided by Goldreich and Ostrovsky [8] and achieves an amortized communication complexity of \( O(\log^3 N) \) data blocks per access and requires only logarithmic storage at the client side.

For read, all levels are searched sequentially for the target block while writes are always performed to the top level. Once a level becomes full, it is reshuffled securely and written to the next level.

The most expensive step of the pyramid ORAM is the reshuffle. Various mechanisms have been proposed to make the reshuffle more efficient. Williams
and Sion [4] show how to achieve an amortized construction with $O(\log^2 N)$ communication complexity with $O(\sqrt{N})$ client storage using an oblivious merge sort. Pinkas et al. [15] use cuckoo hashing and randomized shell sort [9] over the original Goldreich and Ostrovsky solution [8] and achieve an amortized communication complexity of $O(\log^2 N)$ with constant client side storage. Goodrich et al. [10] show how to de-amortize the original square root solution and hierarchical solution [8] and achieve a worst-case complexity of $O(\log N)$ in the presence of $O(n^r)$ client side storage where $r > 0$. In [12], Kushilevitz et al. use cuckoo hashing and rotating buffers to provide a de-amortized construction of the original hierarchical solution [8] which achieves a worst-case communication complexity of $O(\log^2 N / \log \log N)$. Unlike the de-amortization techniques used in [10, 12] where each query performs an additional fixed amount of work for the reshuffle, PD-ORAM [22] provides a way to de-amortize the level construction with parallel clients that perform one (or more) level reshuffles in the background.

**Tree-based ORAM.** In contrast to de-amortized ORAM constructions, tree-based ORAMs are naturally un-amortized (the worst-case query cost is equal to the average cost). A tree-based ORAM organizes the data as a binary (or ternary) tree. Each node of the tree is a bucket which can contain multiple blocks. Usually a block is randomly mapped to a leaf in the tree. The ORAMs maintain the following invariant: a block mapped to a leaf resides in any one of the buckets on the path from the root to the leaf to which the block is mapped. The position map which maps blocks to leaves is either stored on the client ($O(N)$ client storage) or recursively on the server for $O(1)$ client side storage at the cost of $O(\log^2 N)$ increase in communication complexity.

To access a particular block, a client downloads all the buckets (or one element from each bucket) along the path from the root to the leaf to which the block is mapped. Once the block has been read, it is remapped to a new leaf and **evicted** back to the tree. Various eviction procedures have been proposed [7, 16, 17, 19, 20] optimizing either bandwidth or the number of round trips or both.

**Write-only ORAM** Li et al. [13] proposed the first write-only ORAM scheme with an amortized write complexity of $O(B \times \log N)$ where $B$ is the block size of the ORAM and $N$ is the number of blocks in the ORAM. Read complexity is $O(N)$ without $O(N)$ local storage.

Blass et al. [5] designed a constant time write-only ORAM scheme assuming an $O(\log N)$ sized stash stored in memory (HIVE-ORAM). It maps data from a logical address space uniformly randomly to the physical blocks on the underlying device. The position map containing this mapping is then recursively stored in $O(\log N)$ smaller ORAMs, a standard technique introduced in [18]. The recursive technique reduces the logical block access complexity for the position map by storing the position map blocks in logical blocks of smaller sizes. Under this assumption, HIVE-ORAM [5] accesses a constant number of logical blocks at the cost of some overflow that is stored in the in-memory stash.
3 Model

Deployment. Data originating from a logical volume (which may support a filesystem on top) is written encrypted to a storage device (either local disk or network storage).

Adversary. The main write-only ORAM adversary is a multi-snapshot adversary that can access the storage device after one or more write operations.

The adversary may save device snapshots – including device specific information, filesystem metadata and bits stored in each block – and compare these snapshots with the current state in an attempt to learn about the location of the written information. The adversary does not however monitor device reads or accesses to client DRAM in between writes to the main device.

Write access patterns. Access patterns are informally defined as an ordered sequence of logical block writes.

Write traces. Write trace are informally defined as the actual modifications to physical blocks due to the execution of a corresponding access pattern.

Security. SeqORAM uses the security model from [5]. Informally, a write-only ORAM is secure if it provides write access pattern indistinguishability against a computationally bounded adversary. Write access pattern indistinguishability has been formally [5] defined as:

For a device with $N$ physical blocks, given two access patterns of equal length $O_0 = \{a_1, a_2, \ldots, a_i\}$ and $O_1 = \{b_1, b_2, \ldots, b_i\}$ and a random write trace, $W = \{w_1, w_2, \ldots, w_i\}$ with $w_i \in N$, $O_0$ is said to be indistinguishable from $O_1$ iff. there exists a function $\epsilon$, negligible in the security parameter $s$ such that

$$Pr[W|O_0] - Pr[W|O_1] < \epsilon(s)$$

Here $Pr[W|O_i], i = 0, 1$ is the probability that $W$ is the sequence of writes caused to the disk by executing $O_i$.

Minimizing I/O latency. A main goal of SeqORAM is to reduce the latency of access for each ORAM operation and specifically the number of disk seeks per access.

Note that this is orthogonal to the typical ORAM performance metrics which involve just optimizing the bandwidth per access. Optimizing bandwidth is important when ORAMs are deployed in networks but seeks tend to dominate for local storage.

Logical vs. physical complexity. Existing ORAMs define access complexity as the number of logical blocks of data accessed per I/O. This allows optimizations by using logical blocks of smaller size [18, 19]. However, it is important to note that standard off-the-shelf storage block-based storage device can access data only in units of physical blocks (sectors). For example, accessing a 256 byte logical block still entails accessing the corresponding 4KB disk block from the disk. Thus, in the context of locally-deployed block-based devices, it is important to also measure the complexity of the physical block accesses per ORAM I/O.
State-of-the-art. The most bandwidth-efficient write-only ORAM to date is HIVE-ORAM [5]. It has a physical block read complexity (number of physical blocks read) of $O(\log \beta N)$ and a physical write complexity (number of blocks written) of $O(\log^2 \beta N)$ where $\beta = B/|addr|$, $B$ is the physical block size in bytes and $|addr|$ is the size of one physical block address in bytes.

4 Overview and Amortized Construction

We introduce first an amortized construction. In further sections we then show how to deamortize efficiently.

Not unlike pyramid-based ORAMs [21, 22], SeqORAM organizes $N$ data blocks into logical levels, each exponentially larger in size than the previous. Levels are stored sequentially to disk – i.e., level $i$ is written to the disk entirely before level $i + 1$.

Each level is divided into buckets, each of which contains $\beta$ fixed-sized blocks. Level $i$ has $k^i$ buckets, where $k$ is a constant fixed at initialization. The last level has $N/\beta$ buckets. Consequently, the number of logical levels is $\log_k(N/B)$ (Figure 1).

In-memory storage. In addition to disk storage, SeqORAM has memory sufficient to hold a constant $(c)$ number of buckets. This is used for performing in-memory bucket merges during level reconstruction (described further).

Invariant. SeqORAM ensures the following two invariants:

1. All blocks in a level are written to disk in ascending order of their their logical blocks addresses.

2. The most recent version of a block is the first one found when ORAM levels are searched sequentially in increasing order of their level number.

Writes. SeqORAM performs data block writes to an in-memory write queue. The queue is of the size of a bucket. All blocks are encrypted randomized (semantically secure) before being written to the write queue. Once the write queue fills up (after $\beta$ writes), its blocks are sorted on the basis of their logical block addresses and flushed to disk (into the ORAM top level bucket).

Merge. Further, once the write queue fills up and the top level of the ORAM is also full, SeqORAM performs a merge on the buckets in the queue and the top level to create the second level. To this end, first, the blocks in the write queue are sorted on their logical addresses. Then, the (already sorted) top level bucket is read into memory. Finally, the two sorted buckets are merged, their blocks reencrypted and written sequentially to buckets in the second level on disk. Note that we need to read the top level into memory to avoid having to seek on disk during the merge.

In general, once all the buckets in level $i$ have been filled up, they are merged with the blocks already in level $i + 1$. This requires reading one bucket each from level $i$ and level $i + 1$, merging them in memory and writing back the merged
Figure 1: SeqORAM Layout. There are $\log_k(N/B)$ levels. Each level contains $k$ buckets with $\beta$ data blocks. Levels are stored sequentially to disk. Each level has a B+ tree map to quickly map logical IDs to blocks within the level.

More specifically, SeqORAM initializes two bucket-sized queues in memory corresponding to level $i$ and level $i+1$. Then, blocks are read sequentially from the two levels to their respective queues. Once the two queues are full, SeqORAM retrieves a block each from both the queues, compares their logical addresses and writes back the block with the lower logical address to level $i+1$. This continues until $\beta$ block have been written sequentially to level...
Note that if the queues contain duplicates then the block in level \(i\) will be written (as it is more recent) and the block in level \(i + 1\) will be discarded. If at any stage the queues become empty before writing \(\beta\) blocks to level \(i + 1\), then fake blocks (such as containing reencryptions of zero) are written instead. Subsequently, the queues are again filled up using blocks read sequentially from the two corresponding levels and the process is repeated to write the next sorted bucket to level \(i + 1\).

A caveat here is that since the outcome of a merge is not trivially predictable, SeqORAM cannot determine the number of blocks that will be left in each queue after writing \(\beta\) blocks to level \(i + 1\). Thus, the bucket in level \(i + 1\) that will be written next as a result of the merge, may not have been read entirely to the corresponding queue yet. In this case, the remaining data blocks in the bucket will be overwritten as a result of the new writes.

To mitigate this, SeqORAM copies the existing blocks from the bucket that have not been read yet for the merge to the next bucket, before writing blocks from the queue. For example, consider that some blocks in bucket \(b\) have not yet been read to memory for the merge although \(b - 1\) buckets have already been written to level \(i + 1\). Then, before writing new blocks to \(b\), SeqORAM copies the remaining blocks in \(b\) to bucket \(b + 1\) while also reading the contents of \(b + 1\) entirely to memory. For indistinguishability, \(\beta\) blocks (either real or fake) are always written to bucket \(b + 1\) even when no blocks need to be copied which prevents an adversary from learning the number of blocks copied from \(b\) to \(b + 1\). Thus, an adversary only observes two sequentially placed buckets being written together at each step of the merge.

Tracking blocks that haven’t been read for a merge from a level can be achieved trivially by initializing a counter when the level merge begins and incrementing with each block write. Since, blocks are always read sequentially for merges, the counter value will point exactly to the block that needs to be enqueued next.

Note that merging levels in this way does not require reading the entire levels into memory – instead only the two bucket-sized queues and one additional bucket from level \(i + 1\) need to be stored in memory as discussed above. If the level is empty, then the level \(i\) blocks are copied directly since they are already sorted.

The last level – which contains \(N/\beta\) buckets, and can thus store all \(N\) blocks – is organized in a slightly more complex fashion. Blocks there are placed at an offset determined by their logical address, independent of the other blocks. Specifically, a block with logical block address \(j\) is always placed at the \(j^{th}\) offset, located in the \(j/\beta^{th}\) bucket. The presence of other blocks does not affect the location of a block in the last level. This allows bounding the ORAM height. It also provides the opportunity to merge duplicate blocks. Since the last level can contain all \(N\) blocks, blocks that are in the last level can be overwritten by more updated copies in the second to last level, during level merge.

The merge protocol described above ensures invariant 1 straightforwardly since periodic level reconstructions ensure that each level contain blocks in in-
Figure 2: Layout of the map B+ tree on the disk. While $\beta$ leaf nodes are written sequentially to the disk, a parent node is created in memory and updated with the value of the maximum logical block address per leaf node. Once the parent node has $\beta$ entries corresponding to the $\beta$ leaf nodes, it is written to the disk sequentially after the leaf nodes. The tree is built recursively in this bottom-up fashion.

Increasing order of their logical block addresses. For invariant 2, new writes/updated copies of blocks are always written first to the write queue and move down sequentially through the levels due to level reconstructions. Consequently, the most updated copy of a block will be encountered first while searching through the ORAM levels sequentially. As we describe below, reads in SeqORAM do not trigger additional writes as in case of full ORAMs [8] and thus do not affect the invariant.

**Level maps.** To read a block from a particular level, it is necessary to first determine the bucket in which the block resides currently. As described above, each level contains blocks sorted on the logical block ID. To make retrieval of blocks from the sorted levels efficient, SeqORAM maintains a *per-level* map at a pre-allocated location on the disk (Figure 1).

The per level map is stored as a B+ tree. Each node of the B+ tree is stored in a physical block. The keys stored in the internal nodes of the B+ tree are logical block addresses while the values at the leaves are buckets in a level in which the corresponding blocks are currently present. More specifically, each leaf node entry is a tuple $<$ logical address, bucket number $>$ and the entries are sorted by logical address. Fake blocks are represented by sentinel values in the leaf node.

Since, each leaf node is written to an individual physical block, and each
entry in the leaf node is of a fixed size (since logical and physical block address
sizes are fixed), the number of entries that can fit in a leaf node can be precisely
determined. Let $B$ be the physical block size and $|\text{addr}|$ the size of an address
(tuple in the leaf node). Then, sizing a bucket to contain $\beta = B/|\text{addr}|$ blocks
ensures that all these blocks have entries within the same leaf node (of size $B$).

Now, level $i$ in the ORAM has $k^i \times \beta$ blocks. To determine the height of
the map $B+$ tree for the level, note that each leaf of the tree contains $\beta$ tuples.
With each tuple then corresponding to a block, the number of leaves in the $B+$
tree for level $i$ is $k^i$. Consequently, the height of the map $B+$ tree (with fanout
$\beta$) for level $i$ is $\log_\beta(k^i) = O(\log_\beta N)$. For a 1TB disk with 4KB blocks and
64-bit addresses, $\beta = 256$ and $\log_\beta N = 4$.

Map construction. The $B+$ tree map for a level is constructed simultaneously
with the level construction. When a new bucket is written to a level (after a
merge), the corresponding leaf node is of the tree is written as well with the
logical addresses and bucket numbers. Once $\beta$ buckets have been written, a
parent node of the $\beta$ leaf node is created in memory with an entry for the
maximum of the logical block addresses in the $\beta$ leaf nodes (Figure 2). Once
the parent node has $\beta$ entries, the parent node is written to the disk as well.
In this way, the tree is constructed in a bottom-up manner with parent nodes
being completely created in memory before being written to the disk. Note that
this requires $\log_\beta(N)$ blocks of memory in order to create the parent nodes.

Reads. SeqORAM performs reads by searching each level in the ORAM
sequentially for the block using the map $B+$ tree for that level. If the block
address is found in the map for a level, the block is retrieved from that level.

Retrieving a particular block from a level thus requires traversing the map
$B+$ tree for that level according to the required logical block ID and then re-
trieving the block from the bucket. This requires $\log_\beta(k^i) + 1$ block accesses.
Recall that the entire bucket does not have to be retrieved since the location
of a block within the bucket can be determined as the leaf entries are already
sorted on logical block addresses.

Moreover, since the adversary does not monitor reads, reading subsequent
levels after a block has been found at a particular level is not necessary. Recall
that the most updated copy of a block is found first while searching through the
ORAM levels sequentially as ensured by invariant 2. This is in contrast to full
pyramid based ORAM designs where all levels must be visited by the search
(either through real access or dummy accesses) to ensure that an adversary also
monitoring reads does not learn the level where a block has been found. Also
note that unlike full pyramid ORAM designs, the block read is not removed from
the level it has been read from and/or added to the top level of the ORAM.

Amortized access complexity. The write access complexity of the above
described construction can be amortized over the level construction. First, note
that during construction of level $i + 1$, $k^i$ buckets in level $i$ are merged with $k^i$
buckets already in level $i + 1$. Also, during merges all buckets in level $i$ and level
$i + 1$ have to be read exactly once from the disk while $k^{i+1}$ buckets are written
exactly twice (due to copies from previous buckets). Further, constructing the
B+tree map requires writing $2 \times k^i$ blocks. Level construction for level $i$ can thus be performed with $O(k^i)$ accesses.

Since each level is exponentially larger than the previous level, level $i$ is constructed only after $k^i \times \beta$ writes. The amortized write access complexity can then be derived as

$$\sum_{i=0}^{\log_k N/B} \frac{O(k^i)}{k^i \times \beta} = O(\log_k N)$$

Note that reads cannot be amortized similar to writes, since reads do not result in writes to the ORAM and subsequent level reconstructions. Instead, the read access complexity is $O(\log_k N \times \log_\beta N)$ since to read a block, the B+ tree map at each level is traversed to locate the level at which the block exists. In later sections we describe a method to reduce the read access complexity.

**Seek time analysis.** As detailed before, a major metric for evaluation of SeqORAM is the number of seeks per I/O. The following details an analysis of the amortized number of seeks as part of the write operations. Note that since individual writes are made to the in memory write queue, and flushing the write queue triggers level reconstructions, the analysis is based on the seeks during level reconstructions. Writing contents of write queue to the top level fortunately requires only 1 seek since all the contents of the sorted write queue is written sequentially.

Consider the level reconstruction for level $i$. For the merge at least $M > 2 \times \beta$ blocks of memory is required – to store two buckets entirely, and compare them in linear time. With $k^i$ buckets at the $i^{th}$ level, the level overflows $k$ times into the $(i+1)^{th}$ level before the $(i+1)^{th}$ level overflows.

Now, when level $i$ overflows into the $i+1^{th}$ level the $j^{th}$ time with $j < k$, the number of buckets accessed includes reading $k^{i-1}$ buckets in level $i$ that need to be merged and $j \times k^{i-1}$ buckets that are currently occupied in level $i+1$, and writing back $(j+1) \times k^{i-1}$ buckets. To summarize, the number of buckets accessed when level $i$ overflows into level $i+1$ the $j^{th}$ time is $k^{i-1} + jk^{i-1} + (j+1)k^{i-1}$.

Further, buckets within levels are stored sequentially on the disk – thus buckets can be read entirely to the memory sequentially without additional seeks. Consequently, reading a bucket requires only 1 seek.

The number seeks for all level reconstructions amortized over the number of writes can then be calculated as follows –

$$s = \sum_{i=0}^{\log_k N/B} \sum_{j=1}^{k} \frac{k^{i-1} + jk^{i-1} + (j+1)k^{i-1}}{2^{i-1} \beta}$$

This can be finally reduced to

$$s = \frac{2\log_k N / \beta (k+1)}{\beta}$$

An important point to be noted is that the above analysis considers 1 seek per each bucket that is read/written per level reconstruction. This is under the
constraint that only 2 buckets can be merged at the same time with allocated memory, \( M = 2\beta \). A key observation here is that in case multiple buckets could be allocated in memory, then merges could be performed with lesser number of seeks. For instance, if \( k^i \) buckets could be read to memory at the same time, then level \( i \) could be read entirely to memory using 1 disk seek for the merge since all buckets are already sorted within the level. Therefore, with more allocated memory, the amortized number of seeks can be optimized further. In fact, if the memory allocated can hold \( 2 \times c \) buckets (that is \( M = 2 \times \beta \times c \)), the amortized seek time reduces by a factor of \( c \). Thus, the amortized seek time in this case is

\[
s = \frac{2 \log_k (N/\beta) (k+1)}{\beta \times c}
\]

Setting \( \beta > k + 1 \) and \( c = O(\log N) \), we get a constant amortized number of seeks per write. For example, actual memory required to be allocated for the merge with 4KB blocks and 1TB disk is \( M = 2 \times 2(3)(30) = 360 \) 4KB blocks, or around 2MB of memory. This is reasonable for current memory-rich systems with at least 2GB of DRAM to support modern operating systems.

**Comparison with existing work.** In comparison to the state-of-the-art, Hive-ORAM [5] which has a physical write access complexity of \( O(\log^2 N) \), the write access complexity of the construction presented above is \( O(\log N) \), amortized over the number of writes. The average number of seeks for Hive-ORAM [5] is \( O(\log^2 N) \) since each write is to a random location. In contrast, as shown above the amortized number of seeks for the construction described above is a constant in the presence of \( O(\log N) \) memory. Interestingly, Hive-ORAM [5] also requires an \( O(\log N) \) sized in-memory stash to store blocks that could not be written to the disk. The read complexity of Hive-ORAM [5] is \( O(\log \beta N) \).

**Lemma 1.** The amortized SeqORAM construction provides write access pattern indistinguishability.

**Proof ( sketch):** The proof follows straightforwardly by construction. Given two equal length access patterns, \( A = w_1, w_2, \ldots, w_i \) and \( B = x_1, x_2, \ldots, x_i \), encrypted writes are first added to the write queue irrespective the logical addresses. When the write queue is flushed, it triggers level reconstructions independent of the contents of the write queue.

During the level reconstructions, blocks from two buckets are merged together on the basis of their logical addresses. Since reads are not observable, the buckets that are being merged are not known to the adversary. The order in which the merged buckets are written is independent of the result of the merge. More specifically, during a merge for level \( i \), an adversary observes buckets always written in the order of 1 to \( k^i \) irrespective of the contents of the buckets. Since, blocks are written reencrypted during a merge, the semantic security of the encryption scheme provides indistinguishability between the blocks written to a bucket after the merge. Thus, observing write traces for \( i \) subsequent writes, an adversary can only do negligibly better than purely guessing whether \( A \) or \( B \) was executed.
5 Deamortized Construction

Although the amortized construction in Section 4 achieves appreciable performance incentives over [5] by reducing the amortized write access complexity and number of seeks per write, it also suffers from two major drawbacks: i) the read access complexity is higher and ii) the worst case write access complexity is $O(N)$ (required to reconstruct the last level). In general, due to the high worst case costs, amortized constructions perform poorly as part of existing systems optimized. For example, level reconstruction especially for larger levels will require blocking I/O for significant periods of time before the level is created entirely, lest consistency is sacrificed. This results in violating I/O timeout for modern operating systems and significantly degrades I/O performance. To make SeqORAM usable in practice, the goal is to deploy SeqORAM without modifying the other layers in the system. In order to achieve this, first a deamortized version of SeqORAM is presented with the same write access complexity and low average number of seeks per write. Then, we describe a mechanism to reduce the read access complexity.

5.1 Deamortization

Previous work [10, 12, 22] has explored various mechanisms to deamortize full pyramid ORAMs by leveraging extra space. All these mechanisms work by performing a fixed portion of the level construction as part of each query. However, as noted in [22], level construction involves multiple subtasks with different completion times. Thus, achieving deamortization in this way only ensures that each query takes roughly the same amount of time by monitoring progress over subtasks and forcing query progress to be proportional to level construction [22]. In contrast, the idea presented here is to achieve strict deamortization which ensures that each write performs exactly the same amount of work and have identical completion time.

Organization. As in [10, 12], the idea here is to use extra space per level to continue writes while reshuffling. First, instead of using a variable branching factor for the number of buckets in a level, $k$ is fixed at 2 – i.e., level $i$ has $2^i$ buckets. Then, each bucket in a level (except for the last level) is duplicated – the original set of buckets is denoted as generation 0 buckets while the duplicated set is denoted as generation 1 buckets. Finally, the entire level (both generation 0 and generation 1) buckets are duplicated to form a secondary buffer. Level $i$ thus contains two buffers each with $2^{i+1}$ buckets – $2^i$ buckets per generation as shown in Figure 4. The last level is not duplicated and contains only one buffer with $N/\beta$ buckets as in the amortized construction from Section 4.

SeqORAM ensures that each generation is individually sorted. Buckets in generation 0 and the corresponding buckets in generation 1 are merged to form the next level. The buffers are used alternatively for merging levels (denoted as the merge buffer) or for writes from the previous level (denoted as the write buffer). More specifically, when generation 0 and generation 1 buckets in level $i−1$ are merged, the resulting buckets are written to the buckets in the write
Figure 3: Deamortization example for 3 levels. In (a) and (b), buckets in the merge buffer of level 1 are merged to form generation 0 of level 2 while writes from the write queue are 1 to the write buffer in level 1. Once generation 0 in write buffer in level 2 has been written (and the merge in the merge buffer of level 1 has been completed), the buffers are switched. In (c) and (d), the merge in level 0 creates generation 1 of level 2 while writes are performed to the write buffer.

![Deamortization Example Diagram]

Figure 4: Level design for the deamortized construction. Each level has two buffers with two generations. In this example, level \(i\) has two buffers with two generations each with \(2^i\) buckets.

![Level Design Diagram]
**Merge.** Figure 3 shows a demonstrative example of the deamortization technique. At a high level, after writing the contents of the write queue to the top level, one new bucket is written to all other levels. In the example, for level 1, generation 0 and generation 1 buckets from the merge buffer are merged while the write from the write queue is performed on the write buffer (Figure 3(a)). The buckets in the merge buffer are merged to form generation 0 buckets of level 2 (Figure 3(a,b)), thus ensuring that generation 0 is sorted.

The merge in level 1 completes once all buckets in generation 0 of the write buffer in level 2 has been written. Next, the buffers are switched for level 1 (Figure 3(c)). The buckets in the merge buffer now are merged to form generation 0 of level 2 (Figure 3(c,d)). Note that after the switch, the previous merge buffer in level 1 is used for subsequent writes from the write queue. Since, the data in the buffer has already been merged and written to level 2 entirely previously (Figure 3(a,b)), the data can be safely overwritten by the new writes.

Buckets from level 2 are merged similarly to form level 3 (partially shown in Figure 3). Thus, deamortization is performed by merging two buckets from each level to be written to the next level while one bucket is written to the level after merging buckets from the previous level. Similar to the amortized construction if during a merge of two buckets $\beta$ real writes cannot be made to the merged bucket, fake blocks are added instead.

Note that during a merge, there will be duplicate contents in the merge buffer of level $i-1$ and the write buffer of level $i$. More specifically, the write buffer of level $i$ will contain the same blocks as in the merge buffer of level $i-1$, organized differently due to the merge. Only after all the block from the merge buffer in level $i-1$ have been written to level $i$, the buffers will be switched and the blocks in the merge buffer will be overwritten by new writes from level $i-2$. This is an important detail for the construction since this allows blocks to be read from the merge buffer of level $i-1$ even while the blocks are being merged and written to the write buffer of level $i$, which is not entirely formed at this time. In 0 of this, SeqORAM would need to search in both the merge buffer for of level $i-1$ and the write buffer of level $i$ for a block while the merge is not complete.

This is because it is not possible to trivially predict the outcome of a merge of two buckets. For example, it is possible for two buckets from the merge buffer to be merged in a way that results in neither of the buckets being completely empty after the merge. Fortunately, the existence of these duplicate blocks does not violate consistency since during reads, level $i-1$ will be checked for the block before level $i$ and the search will stop once the block is found.

**Merge to last level.** Recall that the last level is not duplicated and contains $N/\beta$ buckets to hold all $N$ blocks. Thus, after merging the second to last level, the blocks are written to last level at offsets determined directly by the logical block addresses. More specifically, as in the amortized construction (Section 4), block with logical address $j$ is written to the $j^{th}$ physical block in the last level. In addition, when the merge is performed, the buckets are still written in the increasing sequential order as detailed below. If a bucket does not contain any
entries as a result of the merge, it is reencrypted and written back. This prevents
an adversary from learning the actual number of empty buckets in the last level
– which leaks the amount of actual data in the ORAM. Indistinguishability
between real modification and reencryption due to semantic security ensures
that the adversary cannot determine whether a bucket is empty.

To achieve this, when the generation 0 and generation 1 buckets in the merge
buffer of the second to last level (each containing $N/2\beta$ buckets) are merged, two
buckets in the last level are written/reencrypted in sequentially increasing order.
This ensures that by the time the $N/2\beta$ buckets in the second to last level are
merged, all $N/\beta$ buckets in the last level have been written/reencrypted. Note
that this trivially ensures that duplicates in the last level are overwritten by
copies in the second to last level.

Correctness. The correctness of the mechanism follows from the fact that
buckets in the merge buffer are completely merged and written to the next level
before the write buffer is full. Note that each buffer has the same number of
buckets and the rate at which buckets are merged and written to the next level
is the same as the rate at which buckets are filled in the write buffer. More
specifically, while one bucket is emptied from the merge buffer of a level, one
bucket is written to the write buffer of the same level.

Reads. Since each generation in a level is individually sorted, level maps need
to be maintained for each generation. Consequently each level has 4 B+ trees –
2 B+ trees per generation in each buffer. Also, while searching for a block in a
level, the trees need to be checked for the most updated location of the block.
The order in which the trees are traversed for a block is as follows: i) generation
1 in the write buffer, ii) generation 0 in the write buffer, iii) generation 1 in the
merge buffer, and iv) generation 0 in the merge buffer.

This order ensures that the most updated copy of the block is found first.
First, if either of the generations in the write buffer is not completely full at a
level, the block will found in the merge buffer of the previous level as discussed
above. If a generation in the write buffer is completely full, then it contains the
most recent copy of the block since the contents of the write buffer have been
written as a result of the last merge from the previous level and the current
merge buffer in that level has already been checked previously. If the block
is not in the write buffer then the merge buffer is checked with generation 1
buckets being checked before generation 0 buckets as they were written more
recently. Thus, although the asymptotic read complexity for this construction is
still $O(\log_\beta N \times \log N)$, the constant increases by a factor of 4 over the amortized
construction in Section 4.

Write access complexity. One bucket is written to each level after the write
queue has been filled. Since the write queue is the same size as the buckets,
the overall write access complexity is $O(\log N)$. Effectively, the deamortization
converts an $O(\log N)$ amortized constructions described in Section 4 with a
worst case of $O(N)$ to an $O(\log N)$ worst case construction.

Number of seek. Note that a bucket is written to each level after the write
queue has been filled. Writing each bucket requires one seek. Thus, $O(\log N)$
Figure 5: Access time map (ATM) traversal in the ORAM levels. Map blocks are placed together with the data blocks in the levels. Once the level is determined corresponding to each node along an ATM path, the level map is used to retrieve the node.

seeks are performed after $\beta$ writes (size of the write queue). If $\beta > O(\log N)$, the number of seeks per write is less than 1. This is a reasonable assumption since for a 1TB disk, $\log N = 30$ and $\beta = 256$ for standard 4KB disk blocks and 64 bit block addresses.

5.2 Efficient Reads

The deanmortized construction described above features a read complexity of $O(\log_3 N \times \log N)$. This is because to find a particular block $\log N$ levels must be checked sequentially until the block is found and checking each level requires traversing the map B+ tree per level with height of $O(\log_3 N)$. Further, in the worst case 4 B+tree needs to be traversed per level as discussed above. This is significantly worse than the $O(\log_3 N)$ read complexity for HIVE-ORAM [5].

The additional read complexity for the construction is the result of checking all $\log N$ levels in the worst case in order to locate a particular block. In contrast, [5] stores a position map to indicate the physical location of each logical block. The position map is stored recursively in $O(\log N)$ smaller ORAMs.

Unfortunately, it is non-trivial to track the precise location of each block in SeqORAM since the physical block where a logical block is currently stored in a level depends on the other blocks present in the sorted levels. However, it is possible to correctly predict the level in which a block is present. The insight here is that ORAM writes trigger level reconstructions deterministically – the number of accesses required to complete merging all buckets in the merge buffer of a level and writing to the write buffer of the next level depends only on the number of buckets in the next level. Further, each flush from the write queue is followed by writing exactly one bucket at each level. So the number of write queue flushes required before the write buffer of a level is full and consequently the merge in the previous level is complete can be precisely determined.

Then, the level in which a block is currently present can be precisely determined by comparing the value of a global counter counting the number of times the write queue has been flushed since the ORAM was initialized and the last
access time for the block (mechanism detailed below). The last access time for each block is the value of the global counter when the block was flushed from the write queue. This mechanism allows checking only one level per block read thus reducing the asymptotic read access complexity by $O(\log N)$.

**Tracking levels for reads.** Consider a block with logical address $x$ that was last accessed when the value of the global access counter was $c$. Also, let the current value of the global counter be $g$. Now, recall that exactly one bucket is written to the write buffer in level $i + 1$ after merging two buckets in level $i$ while one bucket is written to the write buffer of level $i$ from level $i - 1$. Once all buckets in write buffer in level $i$ ($2^{i+1}$ buckets) are full, all the buckets in the merge buffer have been merged and written to a generation in the write buffer of level $i + 1$ ($2^{i+1}$ buckets). Thus, the merge buffer now contains duplicates and can be discarded.

Importantly, this implies that the total occupancy of level $i$ in terms of number of filled buckets (with either real or fake blocks) never exceeds $2^i$ i.e., $2^i$ buckets per each generation. In fact, the number of logically occupied buckets is always maintained at $2^i$ since new writes to the write buffer is performed at the same rate as merges (and subsequently writing to the next level) on the merge buffer. Given $g$ and $c$, it is then possible to determine $i$ such that

$$
\sum_{j=1}^{i-1} 2^{j+1} < g - c < \sum_{j=1}^{i} 2^{j+1}
$$

If $N < g - c$, then $x$ is in the last level. Recall that in the last level $x$ will be at a location determined only by its logical block address and thus can be retrieved trivially.

The above equation determines the number of levels that have been filled with new data after $x$ has been written. Now, $x$ can be in either of the three following buffers: i) $x$ can be in the merge buffer of level $i$, ii) $x$ can be in the write buffer of level $i$ and iii) $x$ can be in the merge buffer of level $i - 1$ if the merge hasn’t completed.

**Identifying buffers.** To identify the correct buffer in which $x$ exists, it is first necessary to determine the number of buckets in the write buffer of level $i$ that was empty when $x$ was written. Since the write buffer of a level fills up periodically as a function of the number of new buckets written (write queue flushes), the number of empty buckets can be calculated as follows. For level $i$ with a write buffer size of $2^{i+1}$ buckets, the write buffer fills up after every $2^{i+1}$ write queue flushes. Thus, when $x$ was written $val = c \% 2^{i+1}$ is the number of buckets already completely filled in the write buffer of level $i$.

Now consider that $r$ buckets were written new to level $i$ after $x$. This can be calculate as

$$
r = g - c - \sum_{j=1}^{i-1} 2^{j+1}.
$$

After $x$ was flushed from the write queue, it subsequently moved down the levels as result of more flushes until it reached level $i$. This required $g - c - r$
write queue flushes. As a result of \( g - c - r \) bucket writes, the number of buckets in the write buffer of level \( i \) that were filled can be determined as

\[
b = (val + g - c - r) \% 2^{i+1}
\]

At this stage, the \( r \) buckets which includes \( x \) were written to the write buffer of level \( i \). Thus, if \( b + r > 2^i \), then \( x \) will in the merge buffer of level \( i \) because the \( r \) bucket writes resulted in the write buffer being full. Otherwise, \( x \) is in the write buffer. Further, if \( b < 2^i \) then \( x \) has been written to generation 0 in the write buffer since this implies that generation 0 was not full when \( x \) was merged and written from the merge buffer in level \( i - 1 \). Otherwise, \( x \) is in generation 1.

Now, when writes to generation 1 in the write buffer of level \( i \) starts, the buckets in the merge buffer for level \( i - 1 \) is discarded/overwritten. The buckets in generation 0 of the write buffer of level \( i \) now contains all buckets that were part of the previous merge buffer in level \( i - 1 \). To determine if generation 0 buckets are full in level \( i \), it is enough to check if \( b + r > 2^i \). If true, \( x \) can be found in generation 0 of the write buffer. Otherwise, \( x \) is still part of the merge buffer in level \( i - 1 \).

To summarize, using the mechanism described above, SeqORAM can correctly predict the level number, the buffer and the generation in which a block \( x \) currently resides. This allows SeqORAM to check only one level.

**Access time map (ATM).** SeqORAM stores an access time map (ATM) for efficiently tracking the last access time of a block. The ATM is stored as a B+ tree (not to be confused with the per level map B+ tree) within the same address space as the data blocks. Specifically, each node of the B+ tree is assigned a logical address within the same address space as the data blocks. Each leaf node of the B+ tree stores a tuple \(< l_{addr}, last\_access\_ctr >\) where \( l_{addr} \) is the logical address of a block and \( last\_access\_ctr \) is the value of the global access counter when the block was last written to the write queue. Recall that the global access counter tracks the number of times the write queue has been flushed since the ORAM initialization. Each leaf node is stored in one disk block. The number of entries that can fit in a disk block depends on the size of the tuple. Assuming 64 bit logical addresses and last access counter values, the number of entries in a block can also be fixed as \( \beta \) (as defined before). Consequently, the height of the tree is \( \log_\beta N \) with a fanout of \( \beta \).

The leaf nodes themselves are then ordered from left to right on the basis of the logical address – the leftmost leaf node has entries for logical block addresses 1 to \( \beta \) while the rightmost leaf node has entries for addresses \( N - \beta \) to \( N \). Thus, it is straightforward to determine the path in the tree corresponding to an entry for a particular block address since the leaf node containing the entry can be determined using the block address only. Each internal node contains a tuple that keeps track of the \(< logical\_address, last\_access\_ctr >\) values of its children nodes. Thus, the tree can be traversed by determining the levels at which the nodes on a path of the tree currently exists using the mechanism described...
Above. Figure 5 shows a schematic diagram of the way in which a path in the map can be traversed in the ORAM using the access counter values.

**Reads.** For a data block read, the corresponding path in the ATM is traversed to determine the level and the buffer at which the block is currently present. For traversing the path, the map B+ tree at $\log_{\beta} N$ levels are searched and the nodes along the path that are currently stored in that level are retrieved. Finally, using the access counter value for the required data block stored at the leaf node, the level at which the block currently exists is determined and the block is retrieved from that level using the map B+ tree for that level. Since the height of the map B+ tree at all levels is $O(\log_{\beta} N)$, the overall read complexity of the construction is $O(\log_{\beta}^2 N)$. Thus, the ATM reduces the read complexity of the deamortized construction from $O(\log_{\beta} N \times \log N)$ to $O(\log_{\beta}^2 N)$ with $\beta >> 2$.

**Updating the ATM.** If a data block is to be updated, then the path in the tree is also updated with the new access time. In this case, the ATM blocks on the path read is updated with the new access counters and are written to the write queue along with the updated data block. Figure 6(a) shows how the nodes along a path in the tree is updated and written after the updated data block in the write queue.

A point to note is that each write of a data block is followed by writing the corresponding path in the ATM that it needs to update. Thus, the actual
number of data blocks that can be written to the write queue before a flush (and consequently the overall write throughput) reduces by a factor equal to the height of the ATM (Figure 6(b)).

Fortunately, exploiting sequentiality in logical blocks written together, allows optimizing the throughput by writing nodes common to the ATM path corresponding to multiple writes only once. For example, in Figure 6(c), if all the data blocks in the write queue have entries within the same leaf node of the ATM, then the updated nodes on the corresponding path can be written only once after all the data writes have been completed. This reduces the overhead of writing the same path multiple times. Since, the height of the tree, $\log_\beta N$ is small (4 for a 1TB database with $\beta = 256$), writing 4 map blocks instead of data blocks in a write queue of size $\beta = 256$ leads to minimal reduction in overall throughput. Thus, for sequential write accesses, SeqORAM achieves higher write throughput than for random write accesses.

**Effect of Caching.** Caching can dramatically improve read throughput by avoiding seeks in between sequential reads. In this case, using a cache of $O(\log_\beta N)$ blocks for storing a path of the ATM allows optimizing the number of nodes that need to be accessed for the next read. If the next read is sequential and has the logical block address within the leaf node in the cache, the ATM traversal to locate the level for the block can be completely avoided. In fact, with purely sequential access, this brings down the overall read complexity to $O(\log_\beta N)$ (to access the B+ tree in the corresponding level) since an $O(\log_\beta N)$ sized ATM path is required to be read only after every $\beta$ accesses (the size of the leaf node in the cache) with $\beta \gg \log_\beta N$.

**Lemma 2.** The deamortized SeqORAM construction provides write access pattern indistinguishability.

**Proof (sketch):** The deamortized construction makes two changes to the amortized construction: an extra buffer is added to each level and the ATM is stored within the ORAM. First, the merging in this case proceeds deterministically and independent of the blocks being merged – buckets in generation 0 of the write buffer of a level are written sequentially before the buckets in generation 1. Once the write buffer is full, it is switched to the merge buffer. Note that buckets are only read from the merge buffer and merged in memory. Since reads are not observable and buckets written after the merge are encrypted with semantic security, the outcome of the merge is not determinable by an adversary.

Further, since reads are not observable, using the ATM during reads to reduce the number of levels searched for a block does not provide any additional information to the adversary. Finally, the ATM blocks are written to the write queue indistinguishably from data blocks due to semantic security of the encryption. 

\[\blacksquare\]
Figure 7: Sequential read and write throughputs for different I/O sizes. Throughputs are in logarithmic scale while I/O size is in MB. Higher is better.

6 Evaluation

Implementation. SeqORAM has been implemented as kernel module device mapper, a Linux based framework for mapping blocks in logical volumes to physical blocks. The default cipher used for encryption is AES-CTR (256 bit) with individual per-block random IVs. IVs are stored in a pre-allocated location on disk. Underlying hardware blocks are 512 bytes each and 8 adjacent hardware blocks constitute a SeqORAM “physical block”. Logical and physical block sizes
Table 1: Throughput comparison in MB/s (higher is better). SeqORAM features a 50x speedup over HIVE [5] for sequential reads and a 100x speedup for sequential writes. SeqORAM random read performance is comparable to HIVE [5].

| Access       | dm-crypt | SeqORAM | HIVE [5] |
|--------------|----------|---------|----------|
| Sequential Read | 91       | 7.14    | 0.135    |
| Sequential Write    | 88       | 1.5     | 0.016    |
| Random Read      | 5.0      | 0.055   | 0.120    |
| Random Write     | 4.3      | 0.020   | 0.014    |

Setup. Benchmarks were conducted on Linux boxes with Intel Core i7-3520M processors running at 2.90GHz and 6GB+ of DDR3 DRAM. The storage device of choice was a 1TB IBM 43W7622 SATA HDD running at 7200 RPM. The average seek time and rotational latency of the disk is 9ms and 4.17ms respectively. The data transfer rate is 300MB/s.

SeqORAM was built on a 256GB physical partition. Benchmarks were performed using FileBench version 1.4.9.1 on Ubuntu 14.04 LTS, kernel version 3.13.6. Results for HIVE-ORAM [5] were collected by compiling the open source project [2] and running benchmarks with the same parameters. All tests were run multiple times and results collected with a 95% confidence interval.

Results. Tests were performed using the sequential and random read/write workload personalities of FileBench. Sequential accesses were measured over a 12GB file by performing individual 1MB sequential IOs. Using a file size twice the available DRAM (6GB here) eliminates caching effects. For random reads/writes, individual I/O sizes were reduced to 4KB. Table 1 compares the sequential and random read/write throughputs for SeqORAM with HIVE-ORAM [5] and dm-crypt, a commonly used Linux device mapper for full disk encryption.

SeqORAM is almost 50x faster over HIVE-ORAM [5] for sequential reads and 100x faster for sequential writes. Random write performance for SeqORAM and HIVE-ORAM [5] are comparable while HIVE-ORAM [5] performs better for random reads as it features a read complexity of $O(\log_3 N)$ compared to $O(\log_2^3 N)$ for SeqORAM. However, as discussed before, modern filesystems ensure locality sensitive data placement and thus rarely need to perform individual random data block reads. Moreover, even for a 1TB disk $\log_3 N = 4$ and therefore the average random read complexity is $\log_2^2 N / 2 = 2 \times \log_3 N$.

Further, Figure 7 compares SeqORAM and HIVE [5] throughputs for different sequential I/O sizes to better understand the effects of sequential accesses. Throughputs are presented in natural log scale. As evident from Figure 7(a), SeqORAM is almost 100x faster for sequential reads over HIVE-ORAM [5] with 8MB sequential I/Os.

A similar speedup is achieved for writes for 1MB sequential I/Os. This corre-
responds to the best case where the write queue contains blocks with sequentially increasing addresses and thus only one path of the level map needs to be updated. As expected, the SeqORAM write throughput plateaus under this best case condition.

7 Conclusion

SeqORAM is a write-only ORAM with write “unlinkability” and 0 locality. SeqORAM is implemented in the Linux kernel. SeqORAM is 100x faster over the state-of-the-art.

References

[1] Adobe software may be snooping through your hard drive right now. Online at http://www.geek.com/apps/adobe-software-may-be-snooping-through-your-hard-drive-right-now-1606145.

[2] Hive. "http://www.onarlioglu.com/hive".

[3] Nsa copycat spyware could be snooping on your hard drive. Online at http://www.techradar.com/news/computing-components/storage/nsa-related-spyware-snooping-on-hard-drives-1285157.

[4] Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008 (2008), The Internet Society.

[5] Blass, E.-O., Mayberry, T., Noubir, G., and Onarlioglu, K. Towards robust hidden volumes using write-only oblivious ram. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (New York, NY, USA, 2014), CCS ’14, ACM, pp. 203–214.

[6] Chakraborti, A., Chen, C., and Sion, R. DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries, 2017.

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

[8] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. Journal of the ACM 43 (1996), 431–473.

[9] Goodrich, M. T. Randomized shellsort: A simple data-oblivious sorting algorithm. J. ACM 58, 6 (Dec. 2011), 27:1–27:26.

[10] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. Oblivious ram simulation with efficient worst-case access overhead. In Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (New York, NY, USA, 2011), CCSW ’11, ACM, pp. 95–100.
[11] Islam, M. S., Kuzu, M., and Kantarcioglu, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In Network and Distributed System Security Symposium (NDSS) (2012).

[12] Kushilevitz, E., Lu, S., and Ostrovsky, R. On the (in)security of hash-based oblivious ram and a new balancing scheme. In Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms (2012), SIAM, pp. 143–156.

[13] Li, L., and Datta, A. Write-only oblivious ram based privacy-preserved access of outsourced data, 2013.

[14] Peters, T., Gondree, M., and Peterson, Z. N. J. DEFY: A deniable, encrypted file system for log-structured storage. In 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014 (2015).

[15] Pinkas, B., and Reinman, T. Oblivious ram revisited. In Proceedings of the 30th Annual Conference on Advances in Cryptology (Berlin, Heidelberg, 2010), CRYPTO’10, Springer-Verlag, pp. 502–519.

[16] Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., and Devadas, S. Constants count: Practical improvements to oblivious ram. In 24th USENIX Security Symposium (USENIX Security 15) (Washington, D.C., Aug. 2015), USENIX Association, pp. 415–430.

[17] Shi, E., Chan, T.-H. H., Stefanov, E., and Li, M. Oblivious ram with $o((\log n)^3)$ worst-case cost. In ASIACRYPT (2011).

[18] Shi, E., Hubert Chan, T., Stefanov, E., and Li, M. Oblivious ram with $o((\log n) 3)$ worst-case cost.

[19] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S. Path oram: An extremely simple oblivious ram protocol. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (New York, NY, USA, 2013), CCS ’13, ACM, pp. 299–310.

[20] Wang, X., Chan, H., and Shi, E. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (New York, NY, USA, 2015), CCS ’15, ACM, pp. 850–861.

[21] Williams, P., Sion, R., and Carbunar, B. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In Proceedings of the 15th ACM Conference on Computer and Communications Security (New York, NY, USA, 2008), CCS ’08, ACM, pp. 139–148.
[22] Williams, P., Sion, R., and Tomescu, A. Privatefs: A parallel oblivious file system. In Proceedings of the 2012 ACM Conference on Computer and Communications Security (New York, NY, USA, 2012), CCS ’12, ACM, pp. 977–988.