Anrin Chakraborti* and Radu Sion

SqORAM: Read-Optimized Sequential Write-Only Oblivious RAM

Abstract: Oblivious RAMs (ORAMs) allow a client to access data from an untrusted storage device without revealing the access patterns. Typically, the ORAM adversary can observe both read and write accesses. Write-only ORAMs target a more practical, *multi-snapshot adversary* only monitoring client *writes* – typical for plausible deniability and censorship-resilient systems.

This allows write-only ORAMs to achieve significantly-better asymptotic performance. However, these apparent gains do not materialize in real deployments primarily due to the random data placement strategies used to break correlations between logical and physical namespaces, a required property for write access privacy. Random access performs poorly on both rotational disks and SSDs (often increasing wear significantly, and interfering with wear-leveling mechanisms).

In this work, we introduce SqORAM, a new locality-preserving write-only ORAM that preserves write access privacy without requiring random data access. Data blocks close to each other in the logical domain land in close proximity on the physical media. Importantly, SqORAM maintains this data locality property over time, significantly increasing read throughput.

A full Linux kernel-level implementation of SqORAM is 100x faster than non locality-preserving solutions for standard workloads and is 60-100% faster than the state-of-the-art for typical file system workloads.

Keywords: Write-Only Access Privacy, Oblivious RAM

DOI 10.2478/popets-2020-0012

Received 2019-05-31; revised 2019-09-15; accepted 2019-09-16.

1 Introduction

Dramatic advances in storage technology have resulted in users storing personal (sensitive) information on portable devices (mobiles, laptops etc). To ensure confidentiality, data can be encrypted at-rest. However, often this is not enough since access sequences of read and written locations leaks significant amounts of information about the data itself, defeating the protection provided by encryption [15]. To mitigate this, one solution is to access items using an oblivious RAM (ORAM) protocol to hide data access patterns from an adversary monitoring the storage device or unsecured RAM. Informally, ORAM protocols ensure (computational) indistinguishability between multiple equal-length query/access sequences.

ORAMs have been typically proposed in the context of securing untrusted remote storage servers e.g., public clouds. However, protecting data access patterns is a key requirement in many other privacy-critical applications including plausibly-deniable storage systems [4, 5, 24], secure data backup mechanisms [2] and secure processors [14]. Plausible deniability ultimately aims to enable users to deny the very existence of sensitive information on storage media when confronted by coercive adversaries e.g., border officers in oppressive regimes. This is essential in the fight against increasing censorship and intrusion into personal privacy [7, 20]

Unfortunately, it is impractical to deploy existing ORAM mechanisms in such systems due to prohibitively-high access latencies deriving from high asymptotic overheads for accessing items and ORAM-inherent randomized access patterns. Also, a full ORAM protocol protecting access patterns of *all* operations in real time may be unnecessary for plausible-deniability. After all, most realistic adversaries in this context only have *multi-snapshot* capabilities and can only access the device periodically while at rest. For example, a border officer in an oppressive regime can inspect a user's device during multiple border crossings and compare disk snapshots. This reveals the changes to the disk due to to user writes but does not compromise privacy of runtime reads [4] (except in the presence of end-point compromise e.g., malware in which case the adversary usually gets full access to the entire system).

This opens up a significant opportunity, namely the idea of a write-only ORAM, a simpler, more effective ORAM that only protects the privacy of write accesses [19]. In fact write-access privacy has been proven to

***Corresponding Author: Anrin Chakraborti:** Stony Brook University, E-mail: anchakrabort@cs.stonybrook.edu
Radu Sion: Stony Brook University, E-mail: sion@cs.stonybrook.edu

be one of the necessary and sufficient requirements for building strong plausible deniability systems [5]. Consequently, write-only ORAMs are a critical component of state-of-the-art plausibly-deniable storage mechanisms including systems such as HIVE [4] and DataLair [5]. Further, it has been shown that write-only ORAMs can achieve significantly better asymptotic performance compared to full ORAMs.

Unfortunately, existing designs are still orders of magnitude slower than the underlying raw media. For example, HIVE is almost *four orders of magnitude slower* than HDDs and *two orders of magnitude slower* than SSDs. The main contributor to this slowdown is the random placement of data meant to break *linkability* between separate writes [7, 24], an important property ensuring that an adversary cannot link a set of writes to each other logically, given multiple snapshots of the media. Random data placement results in dramatically increased disk-seek related latencies. Non-locality of access also interferes with numerous higher-level optimizations including caching policies, read-aheads etc.

To mitigate this, Roche et al. [24] recently proposed a write-only ORAM that preserves locality of access for writes. This is based on the idea that *unlinkability* can also be achieved by writing data to the storage media in a canonical form – e.g., by writing logical data blocks sequentially at increasing physical block addresses, independent of their logical block addresses, similar to a log-structured file system. Unfortunately, this does not solve the problem. Sequentially-written physical blocks rarely translate in locality for the logically-related items. In fact, sequential-write log structured file systems perform quite poorly for *reads* as logically related data ends up scattered across the disk over time [16]. And since reads tend to dominate in modern workloads – e.g., over 70% of ext4 requests are reads [18, 25] – optimizing for logical *reads* is especially important.

This paper introduces the philosophy, design and implementation of SqORAM, a new write-only ORAM that preserves locality of access for *both reads and writes.* SqORAM introduces a seek-optimized data layout: logically-related data is *initially placed* and then *maintained throughout its lifetime* in close proximity on the underlying media, to dramatically increase locality of access and overall throughput, especially when paired with standard file systems.

Locality-Preserving Hierarchical Layout. SqORAM smartly adapts hierarchical ORAM [10] techniques for periodic efficient reshuffles keeping logically-related items in close proximity. Specifically, hierarchical ORAMs store blocks in multiple *levels*, where each level is twice the size of the previous level. Blocks are stored at random locations in a level and the contents in each level are periodically reshuffled and moved to the next level. In SqORAM data is organized similar to hierarchical ORAMs. However, instead of randomized block placement, SqORAM *stores blocks per-level sorted on their logical address.* Related blocks with logically contiguous addresses are stored close together in the levels, and can be fetched efficiently while eliminating seek-related latencies.

A key set of insights underlies this: (i) in standard ORAMs, randomized block placement is mainly necessary to protect the privacy of read patterns, and (ii) storing blocks in a standard canonical form (e.g., sorted on logical address) does not reveal write access patterns.

Asymptotically-Efficient Level Reshuffles. In standard hierarchical ORAMs, level reshuffles are expensive. Random block placement (and read-privacy guarantees) necessitates complex oblivious sorting based mechanisms to securely reshuffle data. Eliminating read-privacy requirements provides the opportunity for simple and asymptotically more efficient (by a factor $\mathbb{O}(\log N)$ for a N block database) level reshuffles.

Efficiently Tracking Blocks. In hierarchical ORAMs locating a particular block on disk is expensive and requires searching in *all* the levels. This is necessary because the location of a block (the level that it is read from) reveals information about its last access time, thus all levels need to be searched to prevent the server from learning the one of interest. SqORAM does not face this requirement and employs a new efficient mechanism to securely and efficiently track the location of blocks based on last access times. To retrieve a block, only one level needs to be searched – this includes an index lookup and reading the block from its current on-disk location.

Evaluation. Compared to randomization-based write-only ORAMs (HIVE [4], DataLair [5]) that have been employed in plausible-deniability schemes, SqORAM is orders of magnitude faster for both sequential reads and writes. Compared to the state-of-the-art [24], SqORAM features a 2x speedup for sequential reads and achieves near raw-disk throughputs in the presence of extra memory. As an application, experiments demonstrate that SqORAM is faster than [24] for a typical file system workload with 70% reads and 30% writes [18, 25] and is only 1.5x slower than the baseline.

2 Related Work

Oblivious RAM (ORAM). ORAMs have been well-researched since the seminal work by Goldreich and Ostrovsky [10]. Most of the work on ORAMs optimize access complexity [10, 11, 11–13, 17, 21, 22, 26–31]. Locality-preserving ORAMs are an emerging field of research [1, 6, 9]. However, full ORAM constructions with locality of access are asymptotically more expensive than traditional ORAMs and in some cases trade-off security for performance. We refer to the vast amount of literature on full ORAM constructions for more details.
Write-Only ORAM. Li and Datta [19] proposed the first write-only ORAM scheme with an amortized write complexity of $\mathbb{O}(B \times \log N)$ where B is the block size of the ORAM and N is the total number of blocks.

Blass et al. [4] designed a constant time write-only ORAM scheme assuming an $\mathbb{O}(\log N)$ sized stash stored in memory. It maps data from a logical address space uniformly randomly to the physical blocks on the underlying device. Chakraborti et al. [5] improved upon the HIVE-ORAM construction by reducing the overall access complexity by a factor of $\mathbb{O}(\log N)$.

Roche et al. [24] recently proposed DetWoORAM, a write-only ORAM that is optimized for sequential writes with $\mathbb{O}(\log N)$ read complexity and $\mathbb{O}(1)$ write complexity. The idea is to write blocks to the disk sequentially, at increasing physical addresses, independent of their logical address, not unlike log-structured file systems. It has been shown [24] that maintaining this layout ensures that a multi-snapshot adversary cannot link a set of writes to each other logically given multiple snapshots of the disk. However, once written to disk, blocks are not guaranteed to remain at the same location and, on updates, blocks are written to new locations (e.g., at the head of the log), thus destroying locality of logical accesses for subsequent reads.

3 Background

Adversary. We consider a multi-snapshot adversary that can observe the storage media not just once but at multiple different times and possibly take snapshots after *every* write operation. The adversary may compare past snapshots including device-specific information, filesystem metadata and bits stored in each block with the current state in an attempt to learn about the location of the written information.

Security Definition. To hide access patterns from a multi-snapshot adversary, a write-only ORAM needs to ensure write-access privacy.

Definition (Write-Access Privacy). *Let* $\vec{y} = (y_1, y_2, \ldots)$ *denote a sequence of operations, where each* y_i *is a* $\mathsf{Write}(a_i, d_i)$*; here,* $a_i \in [0, N)$ *denotes the logical address of the block being written, and* d_i *denotes a block of data being written. For an ORAM scheme* Π*, let* $\mathsf{Access}^{\Pi}(\vec{y})$ *denote the physical access pattern that its access protocol produces for the logical access sequence* $\vec{y}$*. We say the scheme* Π *ensures* write-access privacy *if for any two sequences of operations* $\vec{x}$ *and* $\vec{y}$ *of the same length, it holds*

$$\mathsf{Access}^{\Pi}(\vec{x}) \quad \approx_c \quad \mathsf{Access}^{\Pi}(\vec{y}),$$

where $\approx_c$ *denotes computational indistinguishability (with respect to the security parameter* λ*).*

3.1 Hierarchical ORAM

In this section, we review hierarchical ORAM constructions, which is an important building block of SqORAM.
Organization. Hierarchical ORAMs [10] organize data into *levels*, with each level twice the size of the previous level. Specifically, for a database with N data blocks, the ORAM consists of $\log N$ levels, with level $i \leq \log N$ containing 2^i blocks (including dummy blocks).

At each level, blocks are stored at uniformly random *physical* locations determined by applying level-specific hash functions on logical addresses. In other words, each level storage can be viewed as a hash table of appropriate size [17, 30]. Blocks are always written to the top level first and periodically move down the levels as a result of reshuffles.
Queries. During queries, all levels are searched sequentially for the target block using the level-specific hash functions to determine the exact location of the block in a particular level. When a block is found at a certain level i, *dummy* blocks are read from rest of the levels.

After a block is read from a certain level in the ORAM, it is written re-encrypted to the top level. Once the top level is full, the contents of the top level are securely reshuffled and written to the next level. This mechanism is applied for all levels in the ORAM.
Reshuffles. The most expensive step of hierarchical ORAMs is the level reshuffle. This is because when reshuffling level i, its contents are obliviously sorted on randomly-assigned tags (e.g., logical address hashes) and written to random locations in level $i + 1$. Conse-

quently, reshuffles are expensive not only in terms of the total I/O, but also in the number of seeks.

4 Overview

SqORAM aims to perform locality-preserving reads and writes, while preserving write-access privacy. A good starting point for this is to place logically-related data close-together on disk initially when access are sequential in the logical domain e.g., by writing logically-related data blocks to adjacent physical blocks *sequentially* [24]. In an initial stage, the disk layout resembles an append-only log, where the *next* logical write is performed by writing data to the head of the log.

The next critical task is to maintain the layout as data ends up being scattered across the disk over time [16]. One way to achieve this is by periodically reshuffling data to bring logically-related items in close proximity. Importantly, the frequency of reshuffles and the corresponding access patterns should not leak write-access privacy. SqORAM adopts a physical layout similar to hierarchical ORAMs (Section 3.1) with several key differences. In this section we overview the SqORAM construction and present key insights. Further details are provided in later sections.

4.1 Locality-Preserving Disk Layout

Organization. Similar to the case of hierarchical ORAMs, in SqORAM N data blocks are organized in a "pyramid" with multiple *levels*, each level twice the size of the preceding one. Levels are further subdivided into two identical buffers – a *merge* buffer and a *write* buffer. Their function will be discussed shortly. The buffers comprise multiple logical "buckets", each bucket containing up to β fixed-sized blocks. The level number determines the number of buckets in the buffers: a buffer in level i contains 2^i buckets and overall level i contains $2^{i+1} \times \beta$ blocks in total. The last level buffers contain N/β buckets and can hold all N blocks. The total number of logical levels is $\mathbb{O}(\frac{\log(N/B)}{\log(k)})$.

Insight 1: Locality-Preserving Storage Invariant. To preserve logical domain access locality (e.g., reads by an overlying file system) it is desirable to physically store blocks sorted on their logical addresses. To achieve this, SqORAM replaces level-specific hash tables used in standard hierarchical ORAMs with a layout sorting blocks by logical addresses. This is allowable since write-only privacy does not require storing blocks at random locations as in standard hierarchical ORAMs – since the adversary does not see reads and thus cannot link reads with writes. Before being written to disk, blocks are re-encrypted/randomized.

SqORAM stores levels in their entirety on-disk under the following invariant, which enables seek-efficient level reshuffles (as will be discussed shortly):

Blocks in level buffers are written to disk in ascending order of their logical addresses.

Level Index. To efficiently track the precise location of blocks, SqORAM stores a search index for the merge and write buffer in each level. In a given level, each buffer has its own on-disk B-tree index, named *buffer map*. Each buffer map node is stored in a physical block. Buffer map tree leaf node entries contain tuples of the form $\langle \mathsf{laddr}, \mathsf{paddr} \rangle$, where laddr denotes the logical address of a block and paddr is an offset within the corresponding buffer where the block currently resides. Entries are sorted on laddr.

Each internal node entry corresponds to a child node and is a tuple of the form $\langle \mathsf{child\_addr}, \mathsf{child\_paddr} \rangle$. Specifically, for each child node, child_addr is the value of the *lowest* logical address noted in the child node and child_paddr is the physical address corresponding to the location of the child node on disk.

We remark that the buffer maps allow faster queries than a binary search over the sorted blocks in the buffers. This is because each B-tree node is stored in a disk block which must be read/written as a whole, and the tuples are small in size – 16 bytes each assuming 8 byte logical and physical addresses. Thus, a large number of tuples can be packed into a single node (e.g., 256 entries for 4KB disk blocks), for a B-tree with a large fan out and small depth. Queries can then be performed more efficiently and with less number of seeks than binary search. As we show in Section 5, the B-tree maps can also be constructed in a seek-efficient manner.

4.2 Asymptotically-Efficient Level Reshuffles

For hierarchical ORAM constructions, expensive level reshuffle mechanisms are primarily responsible for high performance overheads. In the SqORAM construction described thus far, level reshuffles would constitute obliviously sorting the combined contents in the merge buffer and the write buffer of a level based on ran-

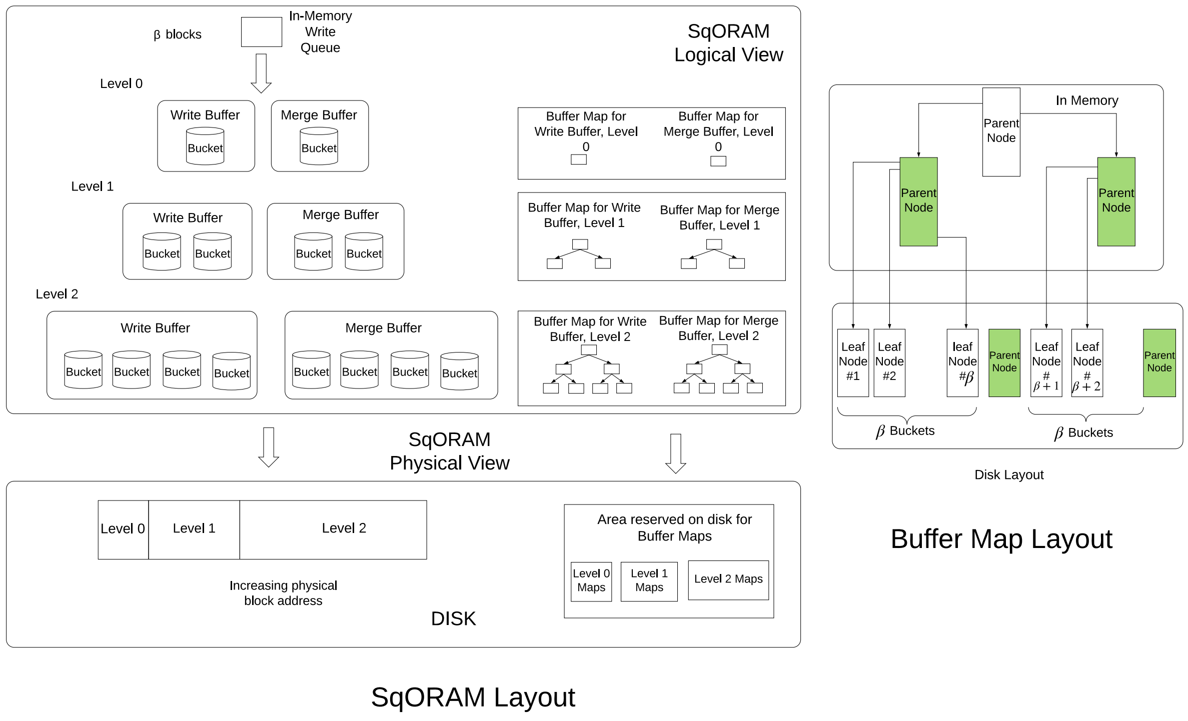


Fig. 1. SqORAM Layout. The database is organized into $\log_k(\frac{N}{B})$ levels. Each level contains two identical buffers. Each buffer in level i contains 2^i buckets with β data blocks. Levels are stored sequentially to disk. Each buffer has a *buffer map* (B-tree) to quickly map logical IDs to blocks within that buffer.

dom tags (e.g., logical address hashes), and writing the sorted contents to the next level. This is not only expensive in terms of access complexity (featuring an asymptotic complexity of $\mathbb{O}(N \log N)$) but also seek-intensive as blocks are read from and written to random locations.

Insight 2: Oblivious Merge for Level Reshuffles. Sorted layouts come with a significant additional benefit: the ability to obliviously and efficiently merge the write and the merge buffers in a level to create the next level during level reshuffles. Obliviously merging is asymptotically faster (by a factor of $\mathbb{O}(\log N)$) than oblivious sorting. Importantly, such oblivious merges are also more seek-efficient. Obliviously sorting blocks on randomly-generated tags requires at least $\mathbb{O}(N \log N)$ seeks. In contrast, obliviously merging blocks from the sorted buffers requires only $\mathbb{O}(N)$ seeks, as the merge can be performed in a single pass. In fact, with a small constant amount of memory, it is possible to reduce seeks further (Section 5).

Worst-Case Construction. Typically, hierarchical ORAMs [10], amortize the cost of the expensive reshuffles over multiple queries. Unfortunately this comes with often prohibitive worst-case delays when clients need to wait (up to hours or days) for reshuffles to complete. This is often impractical for existing system stacks with pre-defined timeouts, such as file systems. Several solutions have suggested de-amortizing the construction [13, 17, 31]. However, as noted in [31], these solutions do not *strictly* de-amortize the level reshuffle, since the subtasks involved in oblivious sorting have widely different completion times. Proper monitoring and *strict* de-amortization of hierarchical ORAMs is a non-trivial task. Benefiting from the oblivious merges, SqORAM presents a naturally un-amortized construction where exactly the same amount of work is done per query, not unlike efficient tree-based ORAM designs [27].

4.3 Efficiently Tracking Blocks for Queries

In hierarchical ORAMs, the exact location of a block is precisely determined by its last access time. Once a block is written to the top level, it moves down the levels according to a precise periodic level reshuffle schedule. Typically, during a query, each of the $\log N$ levels is searched for the matching block. Once found at a particular level, the search continues in the next levels by reading dummy blocks, to hide the location where the block has been found. However, when reads cannot be observed by the adversary, the search can stop as soon as the block is found at a particular level.

Insight 3: Tracking Block Locations Using Last Access Time Based Position Map. Moreover, using the *last access time* of a block, SqORAM can precisely track its location and perform queries efficiently by directly reading the block from the level where it currently resides. Time is measured by a write counter tracking the number of writes since initialization. Last access time information, in conjunction with the current time, allows a precise determination of the level and buffer in which a particular block resides.

The critical challenge is to privately and efficiently store this information. With enough in-memory storage, the last access times can be stored in memory, and synced to the disk on clean power-downs. On power failure or dirty power-downs, the information can be reconstructed in a linear pass over the level indexes only.

However, with limited memory, this information needs to be stored and obliviously queried from disk. To this end, SqORAM maintains an oblivious *access time map* (ATM) structure. The ATM is similar to a B+ tree with one key difference – *instead of each B+ tree node storing physical addresses as pointers to its children, the last access times of the children nodes act as pointers and are stored in the nodes.* ATM nodes are stored in the same ORAM along with the data. The ATM can be traversed from the root to the leaf for determining the location of each child node on the path in the ORAM, based on its last access time value. This is detailed in Section 6.3. ßUsing the ATM, SqORAM can reduce the number of index lookups during queries by a factor of $\mathbb{O}(\log N)$.

5 Amortized Construction

We first introduce an amortized construction to demonstrate our key idea. Later Section 6 shows how to de-amortize efficiently.

Search Invariant. As with most hierarchical ORAM constructions, SqORAM ensures the following invariant:

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.

5.1 SqORAM Operations

In this section, we detail the SqORAM operations.

- write(b, d)*: Writes block with address b and data d.*
- merge(i)*: Merge contents of the buffers in level i and write to level $i+1$ on disk.*
- read(b)*: Read block with address b from the ORAM.*

Writes. SqORAM performs data block writes to an in-memory *write queue*. The queue is of the size of a bucket. When the write queue is full (after β writes), its blocks are sorted on their logical block addresses and flushed to the *write* buffer of the ORAM top level.

Merging Levels: Intuition. Once the contents of the write queue has been written to the write buffer of the top level, SqORAM checks the state of the merge buffer of the top level. Specifically, at this stage the following two cases are possible:

- *Merge buffer is empty:* In this case, the buffers are logically switched – the write buffer becomes the merge buffer and the previously empty merge buffer becomes the write buffer for future accesses.
- *Merge buffer is full:* In this case, the contents of the write buffer and the merge buffer of the top level are merged together to create the write buffer of the second level. To this end, the (sorted) write buffer and merge buffer buckets are read into memory. The two buckets are merged, their blocks re-encrypted and written sequentially to the write buffer buckets in the second level.

Merging Levels: Protocol. Formally, merge(i) (Algorithm 1 in Appendix) includes the following steps:

- *Setup:* Initialize two *bucket-sized* queues in memory corresponding to the write buffer (q_w) and the merge buffer (q_m) of level i.
- *Fill up queues:* Read sequentially from the corresponding buffers to the respective queues until full (Steps 2 - 13). In case all $2^i \cdot \beta$ blocks have already been read from the buffers, *fake* blocks (blocks assigned a logical block address of $N+1$, containing random data) are added to the queues instead.
- *Write blocks from queues to next level until empty:* Retrieve a block *each* from both the queues, compares their logical addresses and writes back the block with the lower logical address *sequentially* to the write buffer in level $i+1$ (Steps 14 - 25).

Handling Duplicates. If the queues contain duplicates then the block in the write buffer queue(q_w) will be written (as it is more recent) and the block in the merge buffer queue (q_m) will be discarded (Steps 21, 22). Also, since *fake* blocks have a logical address of $N+1$, real blocks will be written to level i before *fake*

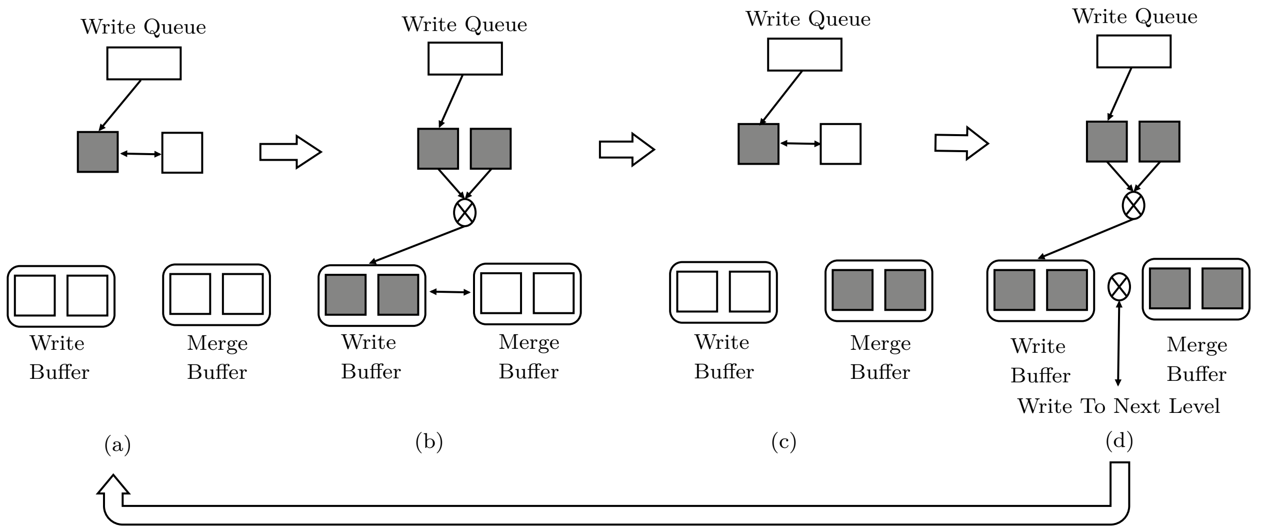


Fig. 2. Illustrative example of the merge protocol. (a) Blocks in the write queue are written to the write buffer in level 0. (b) The write queue is flushed to the empty write buffer in level 0. After this, the blocks in the write buffer and merge buffer of level 0 are merged and written to level 1. Once the write buffer of level 1 is full, the buffers iare switched. Figure (c) and (d) show how the two buffers in level 1 are similarly filled and then merged and written to the next level(s).

blocks. This however is not a security leak since the location of fake and real blocks are important only while performing reads – which is not protected by SqORAM. Semantic security ensures content-indistinguishability.

Bottom-Up Buffer Map Construction. SqORAM employs a novel mechanism to optimize the number of disk seeks that need to be performed to construct a particular B-tree buffer map. Specifically, the B-tree buffer map for the write buffer of a level is constructed in a *bottom up* fashion when blocks are written as a result of a merge (Algorithm 1, Step 24). In particular, after a new bucket is written to the write buffer of a level (Steps 14 - 25, Algorithm 1), a new leaf node is added to the corresponding B-tree buffer map, containing logical and physical addresses of blocks in that bucket.

After β leaf nodes have been added to the B-tree as a result of subsequent accesses, a parent node of the β leaf nodes is created in memory with an entry for the *minimum* of the logical block addresses in the corresponding leaf nodes (Figure 1). Once the parent node has β entries, the parent node is written to the disk as well. In this way, parent nodes are created in memory before being written to the disk next to the children nodes. *Writing parent nodes next to the children nodes sequentially on disk optimizes the number of seeks that need to be performed while constructing the tree.* Note that this requires $\mathbb{O}(\frac{\log N}{\log \beta})$ blocks of memory in order to store the parent nodes up to the root.

Theorem 1. *The amortized merge procedure (Algorithm 1 in Appendix) ensures that all data blocks from the merge and write buffers of level $i-1$ (for any $i \leq logN$) are merged in ascending order of their logical addresses and written to level i, within $2^i \cdot \beta$ accesses.*

Proof. W.l.o.g. consider that at a particular stage of the merge, $x \leq 2^{i-1} \cdot \beta$ blocks have been written from the *write buffer* of level $i - i$ and $y \leq 2^{i-1} \cdot \beta$ blocks have been written from the *merge buffer* of level $i-1$ to level i. Since, none of the buffers have been read entirely ($ctr_w, ctr_m \leq 2^{i-1} \cdot \beta$), both q_w and q_m will contain only real blocks at this stage. Now, consider that in subsequent $2^{i-1} \cdot \beta - y$ accesses, all remaining blocks from the level $i-1$ merge buffer are written to level i (the same argument holds if blocks from the write buffer are written to level i instead). In this case, q_m will contain fake blocks for the remaining $2^{i-1} \cdot \beta - x$ accesses (Step 13).

Since fake blocks invariably have logical address $N+1$ (greater than the logical address of any real data block), the next $2^{i-1} \cdot \beta - x$ writes will be from q_w (Step 23), until both q_w and q_m contain fake blocks. Further, either of the buffers in level $i-1$ can contain at most $2^{i-1} \cdot \beta$ real blocks and fake blocks can be added to q_w only after all real data blocks have been written (Step 7). Thus, the remaining real blocks from the write buffer will be necessarily written to level i within the next $2^{i-1} \cdot \beta - x$ accesses. ■

Theorem 2. *The amortized merge protocol (Algorithm 1 in Appendix) ensures that during the merge all writes to level i are uncorrelated and indistinguishable, independent of the logical block addresses.*

Proof (sketch): Observe that while merging the buffers in levels $i-1$ and writing the constituent blocks to level i (for any $i \leq logN$), the only steps observable to the adversary are the writes performed by Steps 40 and 43 (Algorithm 1). The rest of the steps involve reads or in-memory operations.

Invariably the merge process writes $2^i \cdot \beta$ blocks to level i every $2^i \cdot \beta$ writes by repeatedly executing Steps 40 and 43. Each execution of Step 40 writes an encrypted block to a predetermined location (public information) in a particular level in the ORAM, irrespective of the logical block address and content. Similarly, each execution of Step 43 writes an encrypted block to a predetermined location in the last level, irrespective of the logical address and content. If there are less than $2^i \cdot \beta$ real blocks to write to level i, *fake* blocks are written instead. Semantic security ensures that fake blocks are indistinguishable from real data blocks. Therefore, observing the location, periodicity and content of the writes does not provide any advantage to the adversary in guessing block addresses and contents.

■

Reads. SqORAM reads are similar to queries in hierarchical ORAM constructions. Specifically, reads are performed by searching each level in the ORAM *sequentially* for the required block. A block may reside either in the write buffer or the merge buffer of a particular level at any given time. Therefore, both the buffers in a level must be checked for a block – *SqORAM checks the write buffer first since it contains blocks that have been more recently updated than the blocks in the merge buffer. As a result, the most up-to-date version of a block is found before other (if any) invalid versions.*

Retrieving a particular block requires querying the maps for the write buffer and the merge buffer (in order) at each level, starting from the top level and sequentially querying the levels in increasing order of the level numbers, until an entry for the block is found. Then, the block is read from the corresponding buffer.

Write Access Complexity. Note that during construction of level i, 2^{i-1} buckets each in the write buffer and the merge buffer in level $i-1$ are merged and written to level i. For the merge 2^i buckets in total have to be read from level $i-1$ *exactly once* while 2^i buckets are written *exactly once* to level i. Further, constructing the B-tree map for the write buffer in level i requires writing 2×2^i blocks. Level construction for level i can thus be performed with $\mathbb{O}(2^i)$ accesses.

Since each level is exponentially larger than the previous level, level i is constructed only after $2^i \times \beta$ writes. The amortized write access complexity is:

$$\sum_{i=0}^{\log N/B} \frac{\mathbb{O}(2^i \cdot \beta)}{2^i \times \beta} = \mathbb{O}(\log N)$$

Read Access Complexity. The read access complexity of the amortized construction is $\mathbb{O}(\log N \times \log_\beta N)$ since to read a block, a path in the B-tree buffer maps at each level must be traversed to locate the level at which the block exists. Each buffer in level i has $2^i \times \beta$ blocks. To determine the height of the buffer map B-trees for a level, note that each leaf of the tree contains β tuples. With each tuple then corresponding to a block, the number of leaves in the B-tree for that level is 2^i. Consequently, the height of the B-trees (with fanout β) for level i is $\log_\beta 2^i = \mathbb{O}(\log_\beta N)$. For a 1TB disk with 4KB blocks and 64-bit addresses, $\beta = 256$ and $\log_\beta N = 4$.

Theorem 3 (Seek analysis). *SqORAM requires $\frac{\mathbb{O}(\log N)}{\beta}$ disk seeks, amortized over the number of writes, to perform level reshuffles across $\log N$ levels with $2 \cdot \beta$ blocks of in-memory memory.*

Proof. Consider the process of merging buffers in level $i-1$ (Algorithm 1). First, blocks are read *sequentially* from the merge buffer and the write buffer of level $i-1$ into the in-memory queues, q_m and q_w respectively, until the queues are full. Observe that filling up the queues requires only two disk seeks overall – to place the head at the starting locations of the respective buffers.

After the queues are full, β blocks are written to the write buffer of level i *sequentially*, which requires *one* disk seek. Finally, a leaf node is added to the corresponding buffer B-tree map with entries for block that are written to the bucket, and the parent node(s) of the leaf node are updated in memory. If required, the parent node(s) are flushed to the disk and written *sequentially* after the leaf node. Overall, writing β blocks to the write buffer in level i and updating the B-tree map requires only a constant number of seeks in total.

Since, the write buffer of level i contains 2^i buckets, the total number of seeks for entirely filling up the write buffer is $\mathbb{O}(2^i)$. Also, the write buffer in level i fills up every 2^i writes. Thus, the number seeks for all level reconstructions amortized over the number of writes is:

$$\sum_{i=0}^{\log_k(\frac{N}{B})} \frac{\mathbb{O}(2^i)}{2^i \cdot \beta} = \frac{\mathbb{O}(\log N)}{\beta}$$

■

Observe that for $\beta = \mathbb{O}(\log N)$, SqORAM requires a constant number of disk seeks to perform level reshuffles, amortized over the number of writes. A bucket size of $\mathbb{O}(\log N)$ entails allocating $\mathbb{O}(\log N)$ blocks of memory for storing the queues. This is not impractical – e.g., the actual memory required to be allocated for the queues in order to achieve an amortized number of seeks equal to 1, with 4KB blocks and 1TB disk is $M = 2 \times 4 \times (30) = 240$ blocks, or 960KB.

Theorem 4. *The amortized SqORAM construction provides write-access privacy (Definition 3).*

Proof (sketch): Consider two equal length write access patterns, $\vec{A} = w_1, w_2, \ldots w_i$ and $\vec{B} = x_1, x_2, \ldots x_i$. When either of the access patters are executed, i encrypted blocks are first added to the in-memory write queue irrespective the logical addresses. Once the write queue is full, its contents are written the top level encrypted with semantic security. The top level contents do not leak any information about whether $\vec{A}$ or $\vec{B}$ was executed.

Flushing the write queue will trigger level reshuffles for $k < \log N$ levels. Theorem 2 shows that the writes to the disk while reshuffling any level are uncorrelated to each other and independent of the block addresses and content. Therefore, the writes performed for reshuffling level $j \leq k$ when $\vec{A}$ is indistinguishable from the writes performed when $\vec{B}$ is executed.

Further, level reshuffles are independent of each other and are triggered at periodic intervals determined solely by the number of writes performed (public information). Therefore, by observing the writes to the top level and the writes due to the level reshuffles an adversary can only do negligibly better than purely guessing whether $\vec{A}$ or $\vec{B}$ was executed.

■

6 De-Amortized Construction

The amortized construction achieves appreciable performance incentives over [4] by reducing the amortized write access complexity and number of seeks per write. However it also suffers from two major drawbacks: i) the read access complexity is higher and ii) the worst case write access complexity is $\mathbb{O}(N)$ (for merging and reconstructing the last level). Therefore, to make SqORAM usable in practice, we first present a practical de-amortized version in the following and then describe how to reduce the read access complexity.

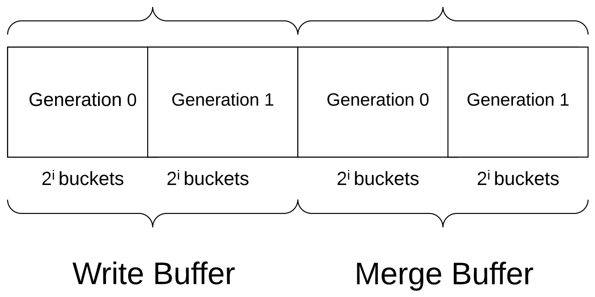


Fig. 3. Level design for the de-amortized construction. Level i has two sets containing 2^i buckets for each buffer. The buckets in the two sets are denoted as *generation 0* buckets and *generation 1* buckets respectively

6.1 De-Amortized Writes

De-amortization for hierarchical ORAMs is achieved by leveraging extra space [13, 17, 31]. These techniques effectively ensure 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 [31]. However, as noted in [31], this does not *strictly* de-amortize the level reshuffle, since subtasks have widely different completion times – correct monitoring and *strict* de-amortization of hierarchical ORAMs is a non-trivial task.

In contrast, our key idea is to leverage the fact that SqORAM does not protect reads and achieve strict *de-amortization.* This ensures that each write performs exactly the same amount of work and has identical completion time, eliminating the need for additional synchronization between queries and reshuffles.

Key Differences from Section 5. In order to de-amortize the construction we make several changes:

- *Leveraging extra space to continue reshuffles in background with queries:* As in [13, 17], SqORAM uses extra space per level to continue writes while reshuffling. In particular, each bucket in a level is duplicated – the two set of buckets are termed *generation 0* and *generation 1* buckets respectively (Figure 3). Each generation is augmented with a B-tree search index (similar to the buffer maps in the amortized construction)
- *Merging contents in generations:* Instead of merging blocks in the merge buffer and the write buffer of a

level, the blocks in the generation 0 and generation 1 buckets of the merge buffer are merged together and written to the write buffer of the next level. The results of the merge are written to a particular generation of the write buffer in the next level:
- If generation 0 buckets are empty, then the blocks after the merge are written to the generation 0 buckets.
- If generation 0 buckets are already full, the blocks after the merge are written to the generation 1 buckets.

Once the write buffer of a level is full, it is switched with the merge buffer. At this stage, the merge buffer is invariably empty. Buffers are used alternatively for merging levels (merge buffer) or for writes from the previous level (write buffer).

Last Level Organization. In addition, the last level is organized differently from the other levels – the last level contains only one buffer with N/β buckets, or N blocks in total. Blocks in the last level are also placed in a different manner – the offset at which a block is placed in the last level buffer is determined by its logical block address – e.g., if the logical block address of a given block is l and the last level buffer starts from physical address x, the block will be placed at physical address $x + l$. Contrast this with other levels where blocks within a buffer are sorted according to the logical address but the physical location has no correspondence with the logical address. As we will show later, this does not leak security and is crucial for the correctness of our de-amortized merge protocol.

De-Amortized Merge: Intuition. *Essentially, the de-amortized merge protocol writes β blocks* sequentially *to a bucket in each of the* $\log N$ *levels.* The specific bucket that is written at a particular level is determined based on the current value of the global access counter. In particular, since the write buffer in level i contains 2^{i+1} buckets in total (two generations with 2^i buckets each) and one new bucket is written each time a merge is executed after a write queue flush, all 2^{i+1} buckets in the write buffer will be written once the write queue has been flushed 2^{i+1}. Subsequently, the write buffer is switched with the empty merge buffer and the next bucket to be written to the new write buffer in level i will be the *first* bucket. *The de-amortized merge protocol reconstructs each level in tandem, one bucket at a time.*

De-Amortized Merge: Protocol. Formally, the de-amortized merge protocol, merge_deamortized (Algorithm 2 in Appendix) includes the following steps:

1. *Setup:* The de-amortized merge protocol requires few supporting counters that are initialized during setup and two queues for each level:
   - q_{x0} – In-memory queue of size β used to store blocks from *generation 0* of the merge buffer of level $x < \log N$.
   - q_{x1} – In-memory queue of size β used to store blocks from *generation 1* of the merge buffer of level $x < \log N$.
   - ctr_{x0} – Tracks the number of blocks that have already been read to q_{x0} from *generation 0* of the merge buffer of level $x < \log N$.
   - ctr_{x1} – Tracks the number of blocks that have already been read to q_{x1} from *generation 0* of the merge buffer of level $x < \log N$.
2. *Merge buffers and write to next level:* For each level $x < \log N$, perform the following sub-steps –
   (a) *Fill Up Queues:* Fill up the two queues q_{x0} and q_{x1} with blocks read *sequentially* from the generation 0 buckets and generation 1 buckets of the merge buffer in level x (Lines 6 - 17). Similar to Algorithm 1, if all real blocks have been read from the two generations, *fake* blocks are added instead (lines 8, 14). The values of ctr_{x0} and ctr_{x1} indicate the next blocks to be read from the respective generations.
   (b) *Write to next level:* Write β blocks *sequentially* to the write buffer in level $x+1$ (Lines 18 - 31).

Last Level Writes. The merge to the last level is handled slightly differently. Recall that due to the special organization of the last level, a block with logical address j must be invariably written to offset j within the last level buffer. Thus, the value of ctr_{next} for the last level is determined keeping in mind that there is only a single buffer with N/β buckets in the last level (Line 5). ctr_{next} points to the next offset in the last level where the next block will be written after the merge. In case this does not match with the logical address of either of the blocks from the two previous level queues, the block at that offset is re-encrypted for indistinguishability (Lines 31 - 32). Otherwise, the required block is written from one of the queues assigned for the second to last level. (Line 34 - 41).

Theorem 5. *The de-amortized merge protocol (Algorithm 2) ensures that by the time the write buffer of level $i < \log N$ is full, all real data blocks from the merge buffer of level i have been written to level $i + 1$.*

Proof. Every successive execution of Algorithm 2 writes β blocks to the write buffer of level i *sequentially.* At the

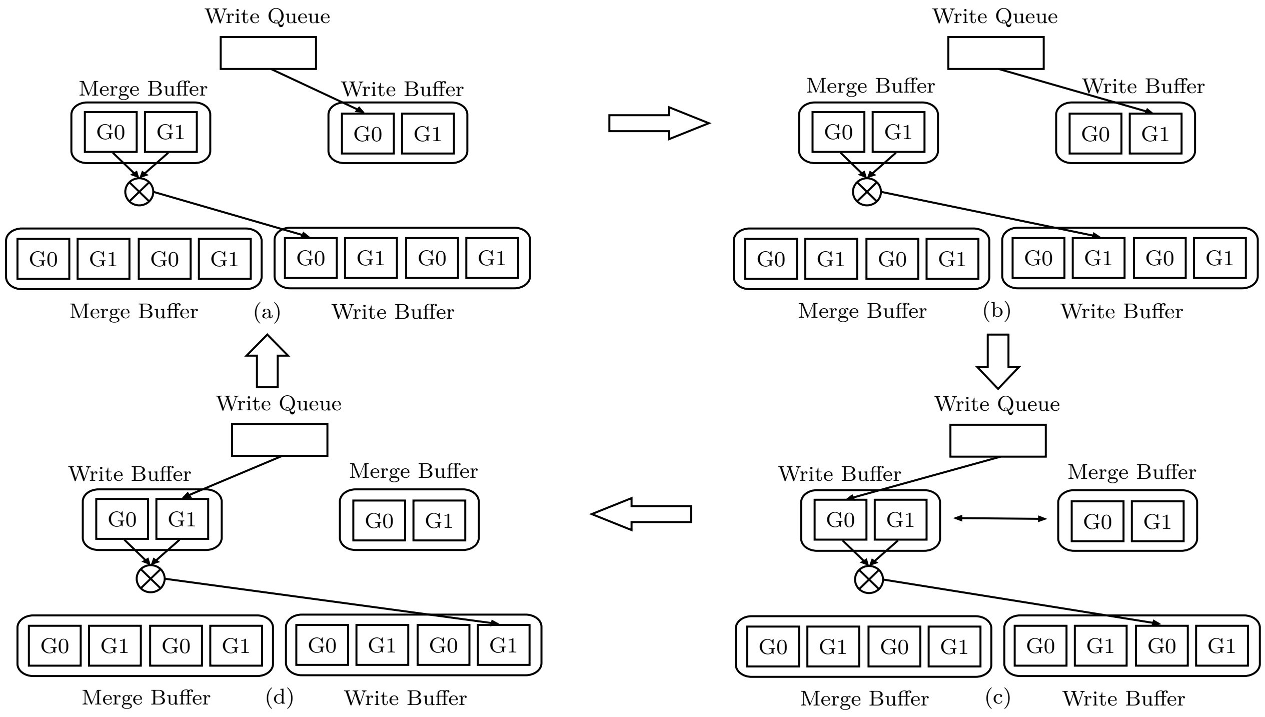


Fig. 4. De-amortization example for 3 levels. In (a) and (b), the generation 0 and generation 1 buckets in the merge buffer of level 1 are merged to form generation 0 of level 2 while writes from the write queue are 1 to the write buffer in level 1. Once generation 0 in write buffer in level 2 has been 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.

same time, β blocks are written from the merge buffer of level i to the write buffer in level $i+1$. Note that within exactly 2^{i+1} successive executions of Algorithm 2, the write buffer of level i will be full. But within the same time, $2^{i+1} \cdot \beta$ blocks will have been written from the merge buffer of level to level $i+1$ (Line 18 - 29). From Theorem 1, all real blocks from the merge buffer of level i will necessarily be written to level $i+1$ within $2^{i+1} \cdot \beta$ writes. Thus, when the buffers are switched after $2^{i+1} \cdot \beta$ writes, all valid content from the merge buffer of level i will be in level $i+1$. ■

Theorem 6. *The de-amortized merge protocol (Algorithm 2 in Appendix) ensures that during the merge all writes to level i are uncorrelated and indistinguishable, independent of the logical block addresses.*

Proof (sketch): Observe that while executing Algorithm 2, the only steps visible to the adversary are the writes performed by Steps 47 and 63. Each execution of Step 47 writes an encrypted block to a predetermined location (public information) in a particular level in the ORAM, irrespective of the logical block address and content. Similarly, each execution of Step 63 writes an encrypted block to a predetermined location in the last level, irrespective of the logical address and content.

Effectively, Steps 33 - 65 write β blocks sequentially in the write buffer in each level, starting from a level-specific predetermined location. The merge protocol is executed invariably after every write queue flush. Further, if there are less than β real data blocks that can be written to a particular level, *fake* blocks are written instead. Semantic security ensures that fake blocks are indistinguishable from real data blocks. Therefore, the locations where blocks are written or the periodicity of the writes does not reveal to the adversary any information about block addresses and contents.

■

Write Access Complexity. For each invocation of the de-amortized merge algorithm after the in-memory write queue has been filled up, one bucket is written to each level. Since the write queue is the same size as the buckets, the overall write access complexity is $\mathbb{O}(\log N)$. Effectively, the de-amortization converts an $\mathbb{O}(\log N)$ amortized construction with a worst case of $\mathbb{O}(\log N)$ to an $\mathbb{O}(\log N)$ worst case construction.

Number of Seeks. Observe that once the write queue is filled up (after β new writes), the constituent blocks are flushed to the ORAM top level and the de-amortized merge protocol (Algorithm 2) is executed. The protocol writes a bucket to *each* of the $\log N$ levels. Writing a bucket to a level requires one seek while filling up the two queues with blocks from the previous level merge buffer requires a seek each. In effect, the write queue flush after β writes triggers a reshuffle mechanism which requires $s = 3 \cdot \log N$ seeks. An additional seek is performed for updating corresponding B-tree buffer map. Thus, the number of seeks performed per write is $\frac{4 \cdot \log N}{\beta}$

Similar to Section 5, with $\beta = \mathbb{O}(\log N)$, the number of seeks performed by the SqORAM de-amortized construction is a constant.

6.2 Efficient Reads

The asymptotic read complexity for the de-amortized construction is $\mathbb{O}(\log_\beta N \times \log N)$. In contrast, position map-based write only ORAMs [4, 5, 24] benefit from asymptotically faster reads with $\mathbb{O}(\log N)$ access complexity. The additional read complexity for hierarchical ORAMs is the result of checking up to $\log N$ levels in order to locate a particular block, which in turn entails querying per-level indexing structures.

Reducing Read Complexity. Unfortunately, it is non-trivial to track the precise location of each block in SqORAM. This is because a block moves down the levels due to periodic reshuffles *even when the block is not specifically updated.* Also, the location of each block in a level depends on other blocks present in that level.

To perform reads efficiently in SqORAM, our key idea is to correctly predict the level, the buffer and the generation in which a block resides currently. Then, only the buffer map for that generation can be queried to determine the actual physical location of the block, thus avoiding buffer map checks in all other levels. This is possible since ORAM writes trigger level reconstructions *deterministically* – 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 depends only on the level size. In particular, the level in which a block currently resides can be accurately predicted by comparing the value of a *global access counter* – tracking the total number of writes since initialization – 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.

Using this mechanism, only *one* level needs to be checked for a block read, reducing the asymptotic read access complexity by a factor of $\mathbb{O}(\log N)$. In fact, the idea can be extended further to also correctly predict the buffer and the generation in the level where a particular block currently resides and reduce the constants further.

First, we describe the mechanism to predict the level, buffer and generation for a a block and then in Section 6.3 show how to efficiently store the last access time information.

Identifying Generation. 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. Thus, after x was written, the write queue was flushed $g - c$ times more. During this time, x moved down the levels due to the merge protocol after every flush.

Predicting the generation to which x will be written in a level is relatively straightforward. Observe that by construction the ORAM is initialized with empty levels – the first time generation 0 of level i will be full is when it contains all blocks written as part of the first $2^i \cdot \beta$ writes. In other words, blocks written during the first $2^i \cdot \beta$ writes (write queue flushed to disk 2^i times) will be written together to generation 0 in the write buffer in level i due to the periodic execution of the merge protocol. Similarly, the next $2^i \cdot \beta$ writes will be to generation 1 in the write buffer in level i. *In particular, the k^{th} group of $2^i \cdot \beta$ writes will be to generation k* mod 2 *in level i write buffer.*

Observation (Identifying Generation)**.** *If a block x is written when $g = c$, and $k = \lfloor c/2^i \rfloor$, then currently x resides in generation j in level i write buffer where $j = k$* mod 2.

Identifying Level. To determine the level in which x currently resides, we specifically track the number of write queue flushes that x spends in level i. Based on this, Algorithm 3 (in Appendix) determines the level by calculating the cumulative time x spent in all levels $j < i$, for each level i (Lines 5 - 10) and comparing with the total number of write queue flushes that have taken place since x was written (Line 4).

Identifying Buffer. To identify the correct buffer in which x currently resides in level i, we need to determine the number of writes that have taken place since x was written to level i. Specifically, the total number of write queue flushes performed after x was written to level i:

$$w = g - c - \sum_{j=1}^{i-1} \mathsf{flush_j} \tag{1}$$

Here $\mathsf{flush}_\mathsf{j}$ is the number of write queue flushes x spend in level j as determined by Algorithm 3. The difference in the number of write queue flushes x cumulatively spent in levels 0 to $i-1$ with the total number of write queue flushes that have been performed since x was written determines the number of write queue flushes performed after x was written to level i. Thus, x is currently in generation 1 in the merge buffer of level i if x was written to generation 0 in level i and $w > 2^i$.

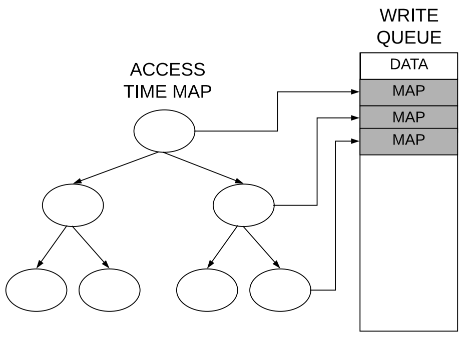


Fig. 5. Writing a path of the ATM in the write queue after updating the path corresponding to an updated data block.

6.3 Access Time Map

To use the mechanism described above, we need to track the last time a particular block was accessed. For this, SqORAM stores an *oblivious data structure* (ODS), named the *access time map* (ATM) within the same ORAM with the data. *In structure, the ATM is effectively a B+ tree but unlike a standard B+ tree where each node stores pointers to its children nodes, each node in the ATM stores an access time value for its children.* The ATM is traversed from the root to the leaf by determining the location of each child node on the path based on its last access counter value as described above.

ATM Design. Each node of the ATM is assigned a logical address within the same address space as the data blocks. In particular, each leaf node of the ATM stores a tuple $\langle \mathsf{laddr}, \mathsf{last\_access\_ctr} \rangle$ where laddr is the logical address of a block and $\mathsf{last\_access\_ctr}$ 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 β (as defined before). Consequently, the height of the tree is $\log_\beta N$ with a fanout of β.

The leaf nodes themselves are ordered from left to right on the basis of the logical address – the leftmost leaf node has entries for logical block addresses 1 to β 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 logical block address uniquely defines the corresponding path in the ATM.

Each internal node contains a tuple that keeps track of the $\langle \mathsf{laddr}, \mathsf{last\_access\_ctr} \rangle$ values of its children nodes. The root of the ATM is stored in-memory and allows traversing a path of the ATM by determining locations of the nodes in the ORAM based on the last access counter values.

Querying the ATM. To read block b (Algorithm 4 in Appendix), first the ATM path corresponding to the leaf containing the entry for b is traversed to determine the last access counter value of the block for b (Lines 1 - 9). This determines the level, buffer and generation where the block currently resides in the ORAM. Then the buffer map of the corresponding level, buffer and generation is queried for the location of b and the block is read from there (Line 11 - 13).

The height of the ATM is $\mathbb{O}(\log_\beta N)$ and the height of the buffer map is bounded by $\mathbb{O}(\log_\beta N)$. Therefore, the overall read complexity is $\mathbb{O}(\log_\beta^2 N)$ – the ATM reduces the read complexity of the de-amortized construction from $\mathbb{O}(\log_\beta N \times \log N)$ to $\mathbb{O}(\log_\beta^2 N)$ with $\beta >> 2$.

Updating the ATM. If a data block is to be written/updated (Algorithm 5 in Appendix), it is first written to the write queue (line 1). This is followed by updating the last accesses counter value for that block in the ATM. Specifically, the leaf node on the ATM path containing an entry for the block is first updated with the new access counter value for the block (Line 4). Then, the path is updated with the new access counter values for the children nodes up to the root (Line 5 - 7), and the updated nodes are added to the write queue (line 6). At this stage, if the write queue is full, it is flushed to the top level and the de-amortized merge protocol (Algorithm 2) is executed (Lines 8 - 10).

Observe that each write of a data block is followed by updating the corresponding path in the ATM. 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).

Sequential Reads. Exploiting sequentiality in logical blocks written together, allows optimizing the through-

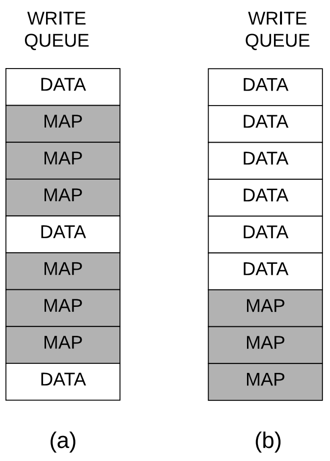


Fig. 6. (a) The state of the write queue when random data blocks are written and the corresponding ATM paths do not intersect. (b) State of the write queue when logically sequential data blocks are written. Only one path common to all the data blocks needs to the be updated for the ATM.

put by writing nodes common to the ATM path corresponding to multiple writes only once. For example, in Figure 6, 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, sequential writes are faster than random writes in SqORAM.

Effect of Caching. Caching can dramatically improve read throughput by avoiding seeks in between sequential reads. In this case, using a cache of $\mathbb{O}(\log_\beta N)$ 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.

Map Caches. To further optimize sequential reads, recently read leaf nodes from the buffer maps can be cached in memory. Since the leaf nodes contain entries sorted on logical addresses, blocks with sequentially increasing logical addresses will have entries within the same leaf node. Thus, for sequential reads, if an entry for a block is found in the ATM path in memory, then instead of querying the buffer map at the required level (and incurring an overhead of $\mathbb{O}(\log_\beta N)$), the entry for that particular block will be found in the cached leaf node for that buffer map.

Theorem 7. *The deamortized SqORAM construction provides write access privacy (Definition 3).*

Proof (sketch): Consider two equal-length write access patterns $\vec{A} = w_1, w_2, \dots w_i$ and $\vec{B} = x_1, x_2, \dots x_i$. First, note that in the de-amortized construction (similar to the amortized construction) when either of $\vec{A}$ or $\vec{B}$ is executed, i blocks are added to the write queue irrespective the logical addresses. Once the write queue is full, its contents are written the top level write buffer, encrypted with semantic security. Further, the de-amortized level reshuffling protocol ensures the same security guarantees as the amortized protocol – while reshuffling level $j < \log N$, the writes to the disk are uncorrelated to each other and independent of the block addresses (Theorem 6). Therefore, by simply observing the writes to the top level and the writes due to the level reshuffles, an adversary can only do negligibly better than purely guessing whether $\vec{A}$ or $\vec{B}$ was executed. ■

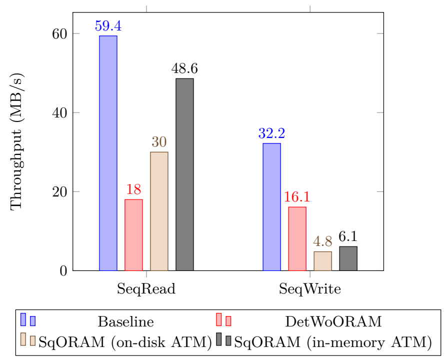
7 Evaluation

SqORAM has been implemented as a kernel device mapper as well as a virtual block device using the Block Device in User Space (BUSE) [8] framework for a fair comparison with all related work.

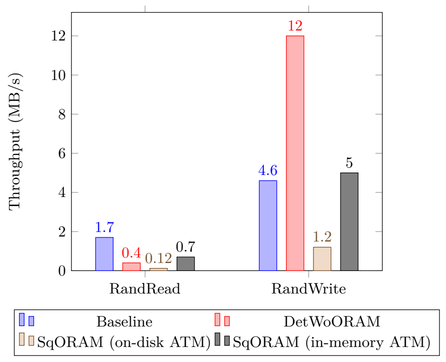
7.1 Kernel-Space Implementation

SqORAM has been implemented as a kernel device mapper and benchmarked in comparison with the kernel implementations of HIVE [4] and DataLair [5]. The cipher used for encryption is AES-CTR (256 bit) with individual per-block random IVs. IVs are stored in a preallocated location on disk. Underlying hardware blocks are 512 bytes each and 8 adjacent hardware blocks constitute a “physical block” (4KB in size).

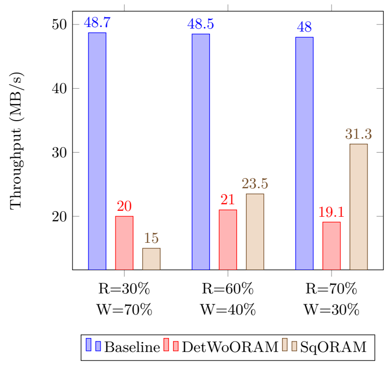
Setup. Benchmarks were conducted on Linux boxes with Intel Core i7-3520M processors running at 2.90GHz and 4GB+ of DDR3 DRAM. The storage device of choice was a 1TB IBM 43W7622 SATA HDD running at 7200 RPM. The average seek time and rotational latency of the disk is 9ms and 4.17ms respectively. The data transfer rate is 300MB/s. SqORAM was built on a



(a) Sequential Access Throughput



(b) Random Access Throughput



(c) Workload Benchmarks

Fig. 7. Throughput comparison in MB/s (higher is better). The baseline is a block device implemented in BUSE that translates FS requests to block requests. (a) When the ATM is stored in memory, SqORAM can achieve almost raw-disk sequential read throughputs and slightly better performance for random reads compared to DetWoORAM. When the ATM is on disk, SqORAM sequential read throughput is higher (almost 2x) than DetWoORAM sequential read throughput. (b) SqORAM outperforms DetWoORAM for random reads. (c) Throughput comparison in MB/s (higher is better) for different read/write distributions. With 70% reads and 30% writes, SqORAM is 1.6x faster than DetWoORAM [24]. DetWoORAM performs generally better for write-intensive workloads.

Table 1. Throughput comparison in MB/s (higher is better). SqORAM features a 150x speedup over HIVE [4] and DataLair [5] for sequential reads and a 100x speedup for sequential writes. SqORAM random read performance is comparable to HIVE [4].

Access	dm-crypt	SqORAM	HIVE [4]	DataLair [5]
Sequential Read	91	21	0.135	0.200
Sequential Write	88	1.5	0.016	0.110
Random Read	5.0	0.055	0.120	0.105
Random Write	4.3	1.0	0.014	0.2

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 [4] were collected by compiling the open source project [3]. We thank the authors of DataLair [5] for providing their implementation. All tests were run multiple times and results were 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 8GB file by performing individual 1MB sequential IOs. Using a file size twice the size of the available DRAM (4GB 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 SqORAM with HIVE, DataLair and dm-crypt, a Linux device mapper for disk encryption.

SqORAM is 150x faster than HIVE [4] for sequential reads and 100x faster for sequential writes. Random write performance for SqORAM and HIVE [4] are comparable while HIVE [4] peforms better for random reads as it features a read complexity of $\mathbb{O}(\log_\beta N)$ compared to $\mathbb{O}(\log^2_\beta N)$ for SqORAM.

7.2 Userland Implementation

To compare with the user space implementation of DetWoORAM [23], SqORAM has also been implemented as a virtual block device using the Block Device in User Space (BUSE) [8] framework. The baseline is a block device which translates file system requests into block requests to the underlying device. It is necessary to build a baseline from the ground up since, as also noted in [24], simply using a loopback device with BUSE as the baseline unfairly overestimates performance by directly offsetting arbitrary lengths within the partition without any address translation to blocks.

Tests were run on a 40 GB ORAM (similar parameters as used in [24]). Each ORAM volume was mounted using an ext4 file system. DetWoORAM was setup with the holding area equal to thrice the size of the main area. Filebench results are presented in Figure 7.

Micro-Benchmark Results. DetWoORAM performs better for writes since it performs two physical writes for each logical write in contrast to the $\log N$ worst case writes in SqORAM. Both logically random and sequential writes in DetWoORAM result in the same physical writes by construction.

Being optimized for reads, SqORAM outperforms DetWoORAM for sequential reads. The advantages of maintaining data locality can be clearly observed in the sequential read throughput, where the overhead compared to the baseline is less than 2x. In fact, for memory-rich systems if the ATM is stored in-memory, SqORAM can achieve sequential read throughputs close to the baseline. For a 40GB partition, the ATM requires 128MB of memory considering 64 bit access time counters. Note that trivially storing in-memory maps for DetWoORAM will not result in similar gains as logically sequential data is not maintained close on disk throughout its lifetime due to frequent updates.

Interestingly, for random writes both DetWoORAM and SqORAM outperform the baseline. This is because for the baseline, the logical address of a block determines the physical address where the data is written on disk. Thus, writing to random logical addresses, incurs a large amount of disk seeks. For both DetWoORAM and SqORAM, the physical addresses for performing writes are not correlated to the logical addresses, and *all* writes are performed while preserving locality of access.

Effect of Workload Distribution. To further investigate the effects of the read/write distribution in workloads on the performance of SqORAM, we evaluated three FileBench workloads with SqORAM and DetWoORAM [24] with different read-write distributions. For the read-intensive workloads ($> 50\%$ reads), SqORAM is almost 1.7x faster than DetWoORAM and only 1.5x slower than the baseline. Note that typical file system workloads are read-intensive [18, 25]. For the write-intensive workload (e.g., for online backup services), DetWoORAM is generally faster than SqORAM.

File System Aging & Effect of Micro-Writes. As noted before, the physical layout of DetWoORAM is similar to a log-structured file system. It is well known that log structured filesystems perform poorly for reads – performance degrades over time as the file system *ages* – as an increasing number of smaller random writes (updates) are performed across large sequential files [16]. This results in a file's blocks being scattered, making sequential reads more expensive. Standard micro benchmarks do not capture this behavior since in most cases, sequential reads are performed on a sequentially written file without interleaving random writes.

To understand the effects of *micro-writes*, we use a sequential read after random write benchmark [16]. The test writes a large file (1GB) sequentially, followed by a sequential read. Then, a fixed number (around 100MB) of random *block-sized* random writes are performed on the file. This is followed by sequentially reading the file again and comparing the read throughput with the earlier reported value. Results are tabulated in Table 2.

The sequential read throughputs for the baseline and SqORAM remain largely unaffected due to data locality – logically-close blocks remain close on disk. DetWoORAM throughput however drops since blocks are increasingly scattered and additional seeks are required.

Table 2. Throughput comparison. (MB/s, higher is better). A series of block-sized random updates to a large file, sequentially-written on disk by DetWoORAM, results in a drop in the sequential read throughput while reading the file subsequently. The drop in throughput for the baseline and SqORAM are not significant.

ORAM	Before random write	After random write
Baseline	61	60.7
SqORAM	47.2	42
DetWoORAM [24]	18.7	10.2

Memory Footprint. SqORAM does not require more than 30MB (mostly for caches and queues), even for 1TB+ ORAMs. The full access time map for a 40GB ORAM takes up less than 128MB, and if stored in memory, the total memory footprint for SqORAM is upper-bound by 200MB. In that case, periodically syncing the ATM with an on-disk copy can ensure crash consistency.

8 Conclusion

This work describes the design and implementation of SqORAM, a write-only ORAM that achieves write access privacy while preserving locality of access for both reads and writes. SqORAM maintains an increased level of data locality over time, thus significantly increasing throughput for both sequential reads and writes.

Compared to randomization-based write-only ORAMs, SqORAM is orders of magnitude faster for both sequential reads and writes. SqORAM is 60% faster than the state-of-the-art for a typical file system workload and is only 1.5x slower than the baseline.

9 Acknowledgements

This research is supported by the National Science Foundation under awards 1526707, 1526102 and by the Office of Naval Research. We thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] G. Asharov, T.-H. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Locality-preserving oblivious ram," vol. 11477, pp. 214–243, 2019.

[2] A. J. Aviv, S. G. Choi, T. Mayberry, and D. S. Roche, "Oblivisync: Practical oblivious file backup and synchronization," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[3] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu, "Hive," "http://www.onarlioglu.com/hive".

[4] E. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu, "Toward robust hidden volumes using write-only oblivious RAM," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 203–214.

[5] A. Chakraborti, C. Chen, and R. Sion, "Datalair: Efficient block storage with plausible deniability against multi-snapshot adversaries," *PoPETs*, vol. 2017, no. 3, p. 179, 2017. [Online]. Available: https://doi.org/10.1515/popets-2017-0035

[6] A. Chakraborti, A. J. Aviv, S. G. Choi, T. Mayberry, D. S. Roche, and R. Sion, "roram: Efficient range ORAM with o(log2 N) locality," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/roram-efficient-range-oram-with-olog2-n-locality/

[7] C. Chen, A. Chakraborti, and R. Sion, "PD-DM: an efficient locality-preserving block device mapper with plausible deniability," *PoPETs*, vol. 2019, no. 1, pp. 153–171, 2019.

[8] A. Cozzette, "Block device in user space (buse)," "https://github.com/acozzette".

[9] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," in *Advances in Cryptology – CRYPTO 2018*, 2018, pp. 371–406.

[10] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, pp. 431–473, 1996.

[11] M. T. Goodrich, "Randomized shellsort: A simple data-oblivious sorting algorithm," *J. ACM*, vol. 58, no. 6, pp. 27:1–27:26, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049697.2049701

[12] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious ram simulation," in *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ser. ICALP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 576–587. [Online]. Available: http://dl.acm.org/citation.cfm?id=2027223.2027282

[13] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Oblivious ram simulation with efficient worst-case access overhead," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '11. New York, NY, USA: ACM, 2011, pp. 95–100. [Online]. Available: http://doi.acm.org/10.1145/2046660.2046680

[14] S. K. Haider and M. van Dijk, "Flat ORAM: A simplified write-only oblivious RAM construction for secure processor architectures," *CoRR*, vol. abs/1611.01571, 2016.

[15] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012. [Online]. Available: https://www.ndss-symposium.org/ndss2012/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation

[16] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, "Betrfs: A right-optimized write-optimized file system," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 301–315. [Online]. Available: http://dl.acm.org/citation.cfm?id=2750482.2750505

[17] E. Kushilevitz, S. Lu, and R. Ostrovsky, "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*. SIAM, 2012, pp. 143–156.

[18] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads," in *USENIX 2008 Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 213–226. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404030

[19] L. Li and A. Datta, "Write-only oblivious ram-based privacy-preserved access of outsourced data," *Int. J. Inf. Secur.*, vol. 16, no. 1, pp. 23–42, Feb. 2017. [Online]. Available: https://doi.org/10.1007/s10207-016-0329-x

[20] T. Peters, M. Gondree, and Z. N. J. Peterson, "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.

[21] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *Proceedings of the 30th Annual Conference on Advances in Cryptology*, ser. CRYPTO'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 502–519. [Online]. Available: http://dl.acm.org/citation.cfm?id=1881412.1881447

[22] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious ram," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 415–430. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-ling

[23] D. S. Roche, A. J. Aviv, S. G. Choi, and T. Mayberry, "Deterministic stash-free write-only oram," "https://github.com/dsroche/detworam".

[24] D. S. Roche, A. Aviv, S. G. Choi, and T. Mayberry, "Deterministic, stash-free write-only oram," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 507–521. [Online]. Available:

http://doi.acm.org/10.1145/3133956.3134051

[25] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '00. Berkeley, CA, USA: USENIX Association, 2000, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267724.1267728

[26] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o((logn)3) worst-case cost," in *ASIACRYPT*, 2011.

[27] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 299–310. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516660

[28] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 850–861. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813634

[29] P. Williams and R. Sion, "Usable PIR," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/08/papers/09_usable_pir.pdf

[30] P. Williams, R. Sion, and B. Carbunar, "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*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 139–148. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455790

[31] P. Williams, R. Sion, and A. Tomescu, "Privatefs: A parallel oblivious file system," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 977–988. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382299

10 Appendix

Algorithm 1: merge(i)

```
$q_w := \phi$ (queue of size $\beta$);
$q_m := \phi$ (queue of size $\beta$);
$ctr_w := 0$ ;
$ctr_m := 0$ ;
$ctr_{next} := 0$;
for $x = 1$ to $2^{i+1}\beta$ do
    while $q_w.notFull$ do
        if $ctr_w \leq 2^i\beta$ then
            Enqueue($q_w$, readFromWriteBuffer($i, ctr_w$));
            $ctr_w = ctr_w + 1$;
        end
        else
            Enqueue($q_w$, fake);
        end
    end
    while $q_m.notFull$ do
        if $ctr_m \leq 2^i\beta$ then
            Enqueue($q_m$, readFromMergeBuffer($i, ctr_m$));
            $ctr_m = ctr_m + 1$ ;
        end
        else
            Enqueue($q_m$, fake);
        end
    end
end
for $y = 1$ to $\beta$ do
    $b_0 = q_0.peek$;
    $b_1 = q_1.peek$;
    if $b_0.add < b_1.add$ then
        $b$ = Dequeue($q_0$);
    end
    else if $b_0.add = b_1.add$ then
        $b$ = Dequeue($q_0$);
        (discard $b_1$ from $q_1$);
    end
    else
        $b$ = Dequeue($q_1$);
    end
    if $i \neq \log N$ then
        writeNextToWriteBuffer($i + 1, ctr_{next}, b$);
    end
    else
        writeNextToWriteBuffer($i, ctr_{next}, b$);
    end
    $ctr_{next} = ctr_{next} + 1$;
end
```

Algorithm 2: merge_deamortized

```
Procedure $Init(\phi)$
    $q_{x0} := \phi$ // Persistent in-memory queue of size $\beta$
      assigned for level $x < \log N$;
    $q_{x1} := \phi$ // Persistent in-memory queue of size $\beta$
      assigned for level $x < \log N$;
    $ctr_{x0} := 0$ // Persistent in-memory counter assigned for
      level $x < \log N$ ;
    $ctr_{x1} := 0$ // Persistent in-memory counter assigned for
      level $x < \log N$;
    $g := 0$ // Global Access Counter (number of times
      write queue has been flushed to the disk);
Procedure Merge($\phi$)
    for $x$ = 1 to $\log N - 1$ do
        if $x \neq \log N - 1$ then
            $ctr_{next} := (g \bmod 2^{x+1}) \times \beta$;
        end
        else
            $ctr_{next} := (g \bmod 2^{x}) \times \beta$;
        end
        while $q_{x0}.notFull$ do
            if $ctr_0 \leq 2^i\beta$ then
                Enqueue($q_{x0}$, readFromMergeBuffer($i, ctr_{x0}$));
                $ctr_{x0} = (ctr_{x0} + 1) \bmod 2^x$;
            end
            else
                Enqueue($q_{x0}, fake$);
            end
        end
        while $q_{x1}.notFull$ do
            if $ctr_{x1} \leq 2^i\beta$ then
                Enqueue($q_{x1}$, readFromMergeBuffer($i, ctr_{x1} +$
                  $2^x \times \beta$));
                $ctr_{x1} = (ctr_{x1} + 1) \bmod 2^x$;
            end
            else
                Enqueue($\mathsf{q}_{\mathsf{x1}}$, fake);
            end
        end
        for $y$ = 1 to $\beta$ do
            $b_0 = q_{x0}.peek$;
            $b_1 = q_{x1}.peek$;
            if $i \neq \log N - 1$ then
                if $b_0.add < b_1.add$ then
                    $b$ = Dequeue($q_{x0}$);
                end
                else if $b_0.add = b_1.add$ then
                    $b$ = Dequeue($q_{x0}$);
                    (discard $b_1$ from $q_{x1}$);
                end
                else
                    $b$ = Dequeue($q_{x1}$);
                end
                writeNextToWriteBuffer($x + 1, ctr_{next}, b$);
            end
            else
                if $b_0.add, b_1.add \neq ctr_{next}$ then
                    Reencrypt($i, ctr_{next}$);
                end
                if $b_0.add < b_1.add$ then
                    $b$ = Dequeue($q_0$);
                end
                else if $b_0.add = b_1.add$ then
                    $b$ = Dequeue($q_0$);
                    (discard $b_1$ from $q_1$);
                end
                else
                    $b$ = Dequeue($q_1$);
                end
                writeToLastLevelBuffer($x + 1, ctr_{next}, b$);
            end
        end
        $ctr_{next} = ctr_{next} + 1$ ;
    end
```

Algorithm 3: Determine_Level(g, c)

```
$\mathsf{flush_i} = 0$ // Number of subsequent write queue flushes $x$
  spent in level $i$ after being flushed to the top level ;
$\mathsf{flush_{sum}} = 0$ ;
$i = 0$;
while $\mathsf{flush_{sum}} \leq g - c$ do
    if $\lfloor c/2^i \rfloor \bmod 2 = 0$ then
        ($x$ was written to generation 0 in level $i$);
        $\mathsf{flush_i} = 3 \times 2^i$;
    end
    else
        ($x$ was written to generation 1 in level $i$);
        $\mathsf{flush_i} = 2 \times 2^i$;
    end
    $i = i + 1$ ;
    $\mathsf{flush_{sum}} = \mathsf{flush_{sum}} + \mathsf{flush_i}$ ;
end
return $i - 1$;
```

Algorithm 4: SqORAM_read(b)

```
$root$ ← Get root node of ATM from memory;
$ATM.path$ ← Get root to leaf path containing entry for $b$;
while not at leaf do
    $child\_num = ATM.path.nextNode.id$;
    $l\_ctr$ ← root.getVal(child_num);
    $child\_level$ ← Determine_Level(g, l_ctr);
    $child$ ← Get Child node from $child\_level$;
    $root = child$;
end
$l\_ctr$ ← root.getVal(b);
$level$ ← Determine_Level(g, l_ctr) ;
$loc\_in\_level$ ← level.bufferMap.query;
$blk$ ← Read block from $loc\_in\_level$;
return $blk$;
```

Algorithm 5: SqORAM_write(b, d)

```
WriteQueue.push(b, d);
$ATM.path$ ← Root to leaf path containing entry for $b$;
$ATM.node = ATM.path.leafNode$;
ATM.node.update(b, g + 1);
while not at root do
    WriteQueue.push(ATM.node.id, ATM.node.data);
    $ATM.node = ATM.path.node.parent$;
end
// Flush contents of Write Queue sequentially to top level
  write buffer;
if $WriteQueue.full$ then
    merge_deamortized()
end
```