arXiv:1809.10559v1 [cs.DC] 27 Sep 2018

Obladi: Oblivious Serializable Transactions in the Cloud

Natacha Crooks[⋆†] *Matthew Burke*[†] *Ethan Cecchetti*[†]
Sitar Harel[†] *Rachit Agarwal*[†] *Lorenzo Alvisi*[†]

[⋆]*University of Texas at Austin* [†]*Cornell University*

Abstract

This paper presents the design and implementation of Obladi, the first system to provide ACID transactions while also hiding access patterns. Obladi uses as its building block oblivious RAM, but turns the demands of supporting transactions into a performance opportunity. By executing transactions within epochs and delaying commit decisions until an epoch ends, Obladi reduces the amortized bandwidth costs of oblivious storage and increases overall system throughput. These performance gains, combined with new oblivious mechanisms for concurrency control and recovery, allow Obladi to execute OLTP workloads with reasonable throughput: it comes within 5× to 12× of a non-oblivious baseline on the TPC-C, SmallBank, and FreeHealth applications. Latency overheads, however, are higher (70× on TPC-C).

1 Introduction

This paper presents Obladi, the first cloud-based key value store that supports transactions while hiding access patterns from cloud providers. Obladi aims to mitigate the fundamental tension between the convenience of offloading data to the cloud, and the significant privacy concerns that doing so creates. On the one hand, cloud services [3, 4, 47, 48, 61] offer clients scalable, reliable IT solutions and present application developers with feature-rich environments (transactional support, stronger consistency guarantees [22, 51], etc.). Medical practices, for instance, increasingly prefer to use cloud-based software to manage electronic health records (EHR) [17, 38]. On the other hand, many applications that could benefit from cloud services store personal data that can reveal sensitive information even when encrypted or anonymized [52, 53, 73, 82]. For example, charts accessed by oncologists can reveal not only whether a patient has cancer, but also, depending on the frequency of accesses (e.g., the frequency of chemotherapy appointments), indicate the cancer's type and severity. Similarly, travel agency websites have been suspected of increasing the price of frequently searched flights [82]. Hiding *access patterns*—that is, hiding not only the content of an object, but also when and how frequently it is accessed, is thus often desirable.

Responding to this challenge, the systems community has taken a fresh look at private data access. Recent solutions, whether based on private information retrieval [2, 30], Oblivious RAM [15, 43, 69], function sharing [82], or trusted hardware [5, 7, 24, 43, 80], show that it is possible to support complex SQL queries without revealing access patterns.

Obladi addresses a complementary issue: supporting ACID transactions while guaranteeing data access privacy. This combination raises unique challenges [5], as concurrency control mechanisms used to enforce isolation, and techniques used to enforce atomicity and durability, all make hiding access patterns more problematic (§3).

Obladi takes as its starting point Oblivious RAM, which provably hides all access patterns. Existing ORAM implementations, however, cannot support transactions. First, they are not fault-tolerant. For security and performance, they often store data in a client-side *stash*; durability requires the stash content to be recoverable after a failure, and preserving privacy demands hiding the stash's size and contents, even during failure recovery. Second, ORAM provides limited or no support for concurrency [12, 69, 74, 85], while transactional systems are expected to sustain highly concurrent loads.

Obladi demonstrates that the demands of supporting transactions can not only be met, but also turned into a performance opportunity. Its key insight is that transactions actually afford more flexibility than the single-value operations supported by previous ORAMs. For example, serializability [60] requires that the effects of transactions be reflected consistently in the state of the database *only after they commit*. Obladi leverages this flexibility to delay committing transactions until the end of fixed-size epochs, buffering their execution at a trusted proxy and enforcing consistency and durability only at epoch boundaries. This

delay improves ORAM throughput without weakening privacy.

The ethos of *delayed visibility* is the core that drives Obladi's design. First, it allows Obladi to implement a multiversioned database atop a single-versioned ORAM, so that read operations proceed without blocking, as with other multiversioned databases [10], and intermediate writes are buffered locally: only the *last* value of any key modified during an epoch is written back to the ORAM. Delaying writes reduces the number of ORAM operations needed to commit a transaction, lowering amortized CPU and bandwidth costs without increasing contention: Obladi's concurrency control ensures that delaying commits does not affect the set of values that transactions executing within the same epoch can observe.

Second, it allows Obladi to securely parallelize Ring ORAM [68], the ORAM construction on which it builds. Obladi pipelines conflicting ORAM operations rather than processing them sequentially, as existing ORAM implementations do. This parallelization, however, is only secure if the write-back phase of the ORAM algorithm is delayed until pre-determined times, namely, epoch boundaries.

Finally, delaying visibility gives Obladi the ability to abort entire epochs in case of failure. Obladi leverages this flexibility, along with the near-deterministic write-back algorithm used by Ring ORAM, to drastically reduce the information that must be logged to guarantee durability and privacy-preserving crash recovery.

The results of a prototype implementation of Obladi are promising. On three applications (TPC-C [79], SmallBank [21], and FreeHealth [41], a real medical application) Obladi is within 5×-12× of the throughput of non-private baselines. Latency is higher (70×), but remains reasonable (in the hundreds of milliseconds).

To summarize, this paper makes three contributions:

1. It presents the design, implementation, and evaluation of the first ACID transactional system that also hides access patterns.
2. It introduces an epoch-based design that leverages the flexibility of transactional workloads to increase overall system throughput and efficiently recover from failures.
3. It provides the first formal security definition of a transactional, crash-prone, and private database. Obladi uses the UC-security framework [14], ensuring that security guarantees hold under concurrency and composition.

Obladi also has several limitations. First, like most ORAMs that regulate the interactions of multiple clients, it relies on a local centralized proxy, which introduces issues of fault-tolerance and scalability. Second, Obladi does not currently support range or complex SQL queries. Addressing the consistency challenge of maintaining oblivious indices [5, 24, 88] in the presence of transactions is a promising avenue for future work.

2 Threat and Failure Model

Obladi's threat and failure assumptions aim to model deployments similar to those of medical practices, where doctors and nurses access medical records through an on-site server, but choose to outsource the integrity and availability of those records to a cloud storage service [17, 38].

Threat Model. Obladi adopts a *trusted proxy* threat model [69, 74, 85]: it assumes multiple mutually-trusting client applications interacting with a single trusted proxy in a single shared administrative domain. The applications issue transactions and the proxy manages their execution, sending read and write requests on their behalf over an asynchronous and unreliable network to an *untrusted storage server*. This server is controlled by an honest-but-curious adversary that can observe and control the timing of communication to and from the proxy, but not the on-site communication between application clients and the proxy. We extend our threat model to a fully malicious adversary in Appendix A. We consider attacks that leak information by exploiting timing channel vulnerabilities in modern processors [13, 35, 42] to be out of scope. Obladi guarantees that the adversary will learn no information about: (*i*) the decision (commit/abort) of any ongoing transaction; (*ii*) the number of operations in an ongoing transaction; (*iii*) the type of requests issued to the server; and (*iv*) the actual data they access. Obladi does not seek to hide the type of application that is currently executing (ex: OLTP vs OLAP).

Failure Model. Obladi assumes cloud storage is reliable, but, unlike previous ORAMs, explicitly considers that both application clients and the proxy may fail. These failures should invalidate neither Obladi's privacy guarantees nor the Durability and Atomicity of transactions.

3 Towards Private Transactions

Many distributed, disk-based commercial database systems [8, 19, 57] separate concurrency control logic from storage management: SQL queries and transactional requests are regulated in a concurrency control unit and are subsequently converted to simple read-write accesses to key-value/file system storage. As ORAMs expose a read-write address space to users, a logical first attempt at implementing oblivious transactions would simply replace the database storage with an arbitrary ORAM. This black-box approach, however, raises both security concerns (§3.1) and performance/functionality issues (§3.2)

Security guarantees can be compromised by simply enforcing the ACID properties. Ensuring Atomicity, Isolation, and Durability imposes additional structure on the

order of individual reads and writes, introducing sources of information leakage [5, 71] that do not exist in non-transactional ORAMs (§3.1). Performance and functionality, on the other hand, are hampered by the inability of current ORAMs to efficiently support highly concurrent loads and guarantee Durability.

3.1 Security for Isolation and Durability

The mechanisms used to guarantee Isolation, Atomicity, and Durability introduce timing correlations that directly leak information about the data accessed by ongoing transactions.

Concurrency Control. Pessimistic concurrency controls like two-phase locking [25] delay operations that would violate serializability: a write operation from transaction T_1 cannot execute concurrently with any operation to the same object from transaction T_2. Such blocking can potentially reveal sensitive information about the data, even when executing on top of a construction that hides access patterns: a sudden drop in throughput could reveal the presence of a deadlock, of a write-heavy transaction blocking the progress of read transactions, or of highly contended items accessed by many concurrent transactions. More aggressive concurrency control schemes like timestamp ordering or multiversioned concurrency control [1, 10, 33, 40, 65, 66, 86] allow transactions to observe the result of the writes of other ongoing transactions. These schemes improve performance in contended workloads, but introduce the potential for *cascading aborts*: if a transaction aborts, all transactions that observed its write must also abort. If a write-heavy transaction T_{heavy} aborts, it may cause a large number of transactions to rollback, again revealing information about T_{heavy} and, perhaps more problematically, about the set of objects that T_{heavy} accessed.

Failure Recovery. When recovering from failure, Durability requires preserving the effects of committed transactions, while Atomicity demands removing any changes caused by partially-executed transactions. Most commercial systems [49, 57, 58] preserve these properties through variants of *undo* and *redo* logging. To guarantee Durability, write and commit operations are written to a redo log that is replayed after a failure. To guarantee Atomicity, writes performed by partially-executed transactions are *undone* via an *undo log*, restoring objects to their last committed state. Unfortunately, this undo process can leak information: the number of undo operations reveals the existence of ongoing transactions, their length, and the number of operations that they performed.

3.2 Performance/functionality limitations

Current ORAMs align poorly with the need of modern OLTP workloads, which must support large numbers of concurrent requests; in contrast, most ORAMs admit little to no concurrency [12, 69, 74, 85] (we benchmark the performance of sequential Ring ORAM in Figure 10a).

More problematically, ORAMs provide no support for fault-tolerance. Adding support for Durability presents two main challenges. First, most ORAMs require the use of a *stash* that temporarily buffers objects at the client and requires that these objects be written out to server storage in very specific ways (as we describe further in §4). This process aligns poorly with guaranteeing Durability for transactions. Consider for example a transaction T_1 that reads the version of object x written by T_2 and then writes object y. To recover the database to a consistent state, the update to x should be flushed to cloud storage before the update to y. It may however not be possible to *securely* flush x from the stash before y. Second, ORAMs store metadata at the client to ensure that cloud storage observes a request pattern that is independent of past and currently executing operations. As we show in §8, recovering this metadata after a failure can lead to duplicate accesses that leak information.

3.3 Introducing Obladi

These challenges motivate the need to co-design the transactional and recovery logic with the underlying ORAM data structure. The design should satisfy three goals: (*i*) security—the system should not leak access patterns; (*ii*) correctness—Obladi should guarantee that transactions are serializable; and (*iii*) performance—Obladi should scale with the number of clients. The principle of *workload independence* underpins Obladi's security: the sequence of requests sent to cloud storage shoud remain independent of the type, number, and access set of the transactions being executed. Intuitively, we want Obladi's sequence of accesses to cloud storage to be statistically indistinguishable from a sequence that can be generated by an Obladi *simulator* with no knowledge of the actual transactions being run by Obladi. If this condition holds, then observing Obladi's accesses cannot reveal to the adversary any information about Obladi's workload. We formalize this intuition in our security definition in §9.

Much of Obladi's novelty lies not in developing new concurrency control or recovery mechanisms, but in identifying what standard database techniques can be leveraged to lower the costs of ORAM while retaining security, and what techniques instead subtly break obliviousness.

To preserve workload independence while guaranteeing good performance in the presence of concurrent requests, Obladi centers its design around the notion of *delayed visibility*. Delayed visibility leverages the observation that, on the one hand, ACID consistency and Durability apply only when transactions commit, and, on the other, commit operations can be delayed. Obladi leverages this flexibility to delay commit operations until the end of *fixed-size epochs*. This approach allows Obladi to (*i*) amortize the

cost of accessing an ORAM over many concurrently executing requests; (*ii*) recover efficiently from failures; and (*iii*) preserve workload independence: the epochs' deterministic structure allows Obladi to decouple its externally observable behavior from the specifics of the transactions being executed.

4 Background

Oblivious **R**emote **A**ccess **M**emory is a cryptographic protocol that allows clients to access data outsourced to an untrusted server without revealing what is being accessed [28]; it generates a sequence of accesses to the server that is completely *independent* of the operations issued by the client. We focus specifically on *tree-based* ORAMs, whose constructions are more efficiently implementable in real systems: to date, they have been implemented in hardware [26, 45] and as the basis for blockchain ledgers [15] with reasonable overheads. Most tree-based ORAMs follow a similar structure: objects (usually key-value pairs) are mapped to a random leaf (or *path*) in a binary tree and physically reside (encrypted) in some tree node (or *bucket*) along that path. Objects are logically removed from the tree and remapped to a new random path when accessed. These objects are eventually flushed back to storage (according to their new path) as part of an *eviction* phase. Through careful scheduling, this write-back phase does not reveal the new location of the objects; objects that cannot be flushed are kept in a small client-side *stash*.

Ring ORAM. Obladi builds upon Ring ORAM [68], a tree-based ORAM with two appealing properties: a constant stash size and a fully deterministic eviction phase. Obladi leverages these features for efficient failure recovery.

As shown in Figure 1, server storage in Ring ORAM consists of a binary tree of *buckets*, each with a fixed number $Z+S$ of *slots*. Of these, Z are reserved for storing actual encrypted data (*real objects*); the remaining S exclusively store *dummy objects*. Dummy objects are blocks of encrypted but meaningless data that appear indistinguishable from real objects; their presence in each bucket prevent the server from learning how many real objects the bucket contains and which slots contains them. A random permutation (stored at the client) determines the location of dummy slots. In Figure 1, the root bucket contains a real slot followed by two dummy slots; the real slot contains the data object a; its left child bucket instead contains dummy slots in positions one and three, and an empty real slot in second position.

Client storage, on the other hand, is limited to (*i*) a constant sized *stash*, which temporarily buffers objects that have yet to be replaced into the tree and, unlike a simple cache, is essential to Ring ORAM's security guarantees; (*ii*) the set of current *permutations*, which identify the

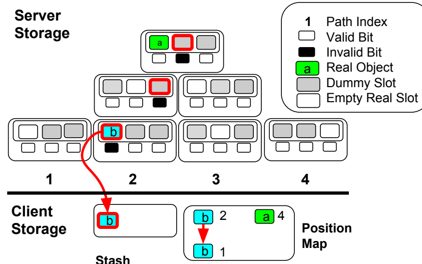


Figure 1: Ring ORAM - Read (Z=1, S=2)

role of each slot in each bucket and record which slot have already been accessed (and marked *invalid)*; and (*iii*) a *position map*, which records the random leaf (or *path*) associated with every data object. In Ring ORAM, objects are mapped to individual leaves of the tree but can be placed in any one of the buckets along the path from the root to that leaf. For instance, object a in Figure 1 is mapped to *path 4* but stored in the root bucket, while object b is mapped to *path 2* and stored in the leaf bucket of this path.

Ring ORAM maintains two core invariants. First, each data object is mapped to a new leaf chosen uniformly at random after every access, and is stored either in the stash, or in a bucket on the path from the tree's root to that leaf (**path invariant**). Second, the physical positions of the $Z+S$ dummy and real objects in each bucket are randomly permuted with respect to all past and future writes to that bucket (i.e., no slot can be accessed more than once between permutations) (**bucket invariant**). The server never learns whether the client accesses a real or a dummy object in the bucket, so the exact position of the object along that path is never revealed.

Intuitively, the path invariant removes any correlation between two accesses to the same object (each access will access independent random paths), while the bucket invariant prevents the server from learning when an object was last accessed (the server cannot distinguish an access to a real slot from a dummy slot). Together, these invariants ensure that, regardless of the data or type of operation, all access patterns will look indistinguishable from a random set of leaves and slots in buckets.

Access Phase. The procedures for read and write requests is identical. To access an object o, the client first looks up o's path in the position map, and then reads one object from each bucket along that path. It reads o from the bucket in which it resides and a valid dummy object from each other bucket, identified using its local permutation map. Finally, o is remapped to a new path, updated to a new value (if the request was a write), and added to the stash; importantly, o is not immediately written back out to cloud storage.

Figure 1 illustrates the steps involved in reading an object b, initially mapped to path 2. The client reads a

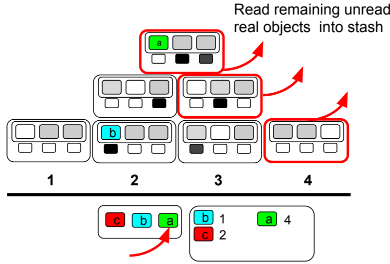


Figure 2: Eviction - Read Phase

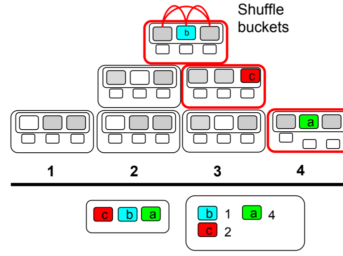


Figure 3: Eviction - Write Phase

dummy object from the first two buckets in the path (at slots two and three respectively), and reads b from the first slot of the bottom bucket. The three slots accessed by the client are then marked as invalid in their respective buckets, and b is remapped to path 1. To write a new object c, the client would have to read three valid dummy objects from a random path, place c in the stash, and remap it to a new path.

Access Security. Remapping objects to independent random paths prevents the server from detecting repeated accesses to data, while placing objects in the stash prevents the server from learning the new path. Marking read slots as invalid forces every bucket access to read from a distinct slot (each selected according to the random permutation). The server consequently observes uniformly distributed accesses (without repetition) independently of the contents of the bucket. This lack of correlation, combined with the inability to distinguish real slots from dummy slots, ensures that the server does not learn if or when a real object is accessed. Accessing dummy slots from buckets not containing the target object (rather than real slots), on the other hand, is necessary for efficiency: in combination with Ring ORAM's *eviction phase* (discussed next) it lets the stash size remain constant by preventing multiple real objects from being addded to the stash on a single access.

Eviction Phase and Reshuffling. The aforementioned protocol falls short in two ways. First, if objects are placed in the stash after each access, the stash will grow unbounded. Second, all slots will eventually be marked as invalid. Ring ORAM sidesteps these issues through two complementary processes: *eviction* and *bucket reshuffling*. Every A accesses, the *evict path* operation evicts objects from the client stash to cloud storage. It deterministically selects a target path, flushes as much data as possible, and permutes each bucket in the path, revalidating any invalid slots. Evict path consists of a read and write phase. In the read phase, it retrieves Z objects from each bucket in the path: all remaining valid real objects, plus enough valid dummies to reach a total of Z objects read. In the write phase, it places each stashed object—including those read by the read phase—to the deepest bucket on the target path that intersects with the object's assigned path. Evict path then permutes the real and dummy values in each bucket along the target path, marking their slots as *valid*, and writes their contents to server storage. Figure 2 and 3 show the evict path procedure applied to path 4. In the read phase, evict path reads the unread object a from the root node and dummies from other buckets on the path. In the write phase (Fig. 3), a is flushed to leaf 4, as its path intersects completely with the target path. Finally, we note that randomness may cause a bucket to contain only invalid slots before its path is evicted, rendering it effectively unaccessible. When this happens, Ring ORAM restores access to the bucket by performing an *early reshuffle* operation that executes the read phase and write phase of evict path only for the target bucket.

Eviction Security. The read phase leaks no information about the contents of a given bucket. It systematically reads exactly Z valid objects from the bucket, selecting the valid real objects from the z real objects in the bucket, padding the remaining $Z-z$ required reads with a random subset of the S dummy blocks. The random permutation and randomized encryption ensure that the server learns no information about how many real objects exist, and how many have been accessed. Similarly, the write phase hides the values and locations of objects written. At every bucket, the storage server observes only a newly encrypted and permuted set of objects, eliminating any correlation between past and future accesses to that bucket. Together, the read and write phases ensure that no slot is accessed more than once between reshuffles, guaranteeing the bucket invariant.

Similarly, the eviction process leaks no information about the paths of the newly evicted objects: since all paths intersect at the root and the server cannot infer the contents of any individual bucket, any object in the stash may be flushed during *any* evict path. Moreover, since all paths intersect at the root, any object in the stash may be flushed during *any* evict path.

5 System Architecture

Obladi, like most privacy-preserving systems [69, 75, 85] consists of a centralized trusted component, the *proxy*, that communicates with a fault-tolerant but untrusted entity, *cloud storage* (Figure 4). The proxy handles concurrency

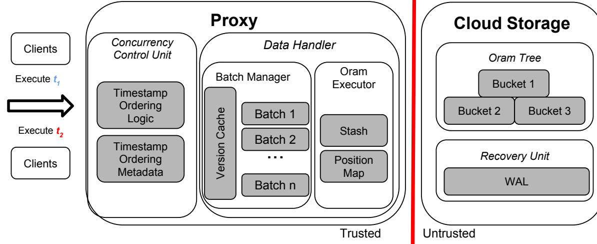


Figure 4: System Architecture

control, while the untrusted cloud storage stores the private data. Obladi ensures that requests made by the proxy to the cloud storage over the untrusted network do not leak information. We assume that the proxy can crash and that when it does so, its state is lost. This two-tier design allows applications to run a lightweight proxy locally and delegate the complexity of fault-tolerance to cloud storage.

The proxy has two components: (*i*) a *concurrency control unit* and (*ii*) a *data manager* comprised of a *batch manager* and an *ORAM executor*. The batch manager periodically schedules fixed-size batches of client operations that the ORAM executor then executes on a parallel version of Ring ORAM's algorithm. The executor accesses one of two units located on server storage: *the ORAM tree*, which stores the actual data blocks of the ORAM; and *the recovery unit*, which logs all non-deterministic accesses to the ORAM to a write-ahead log [50] to enable secure failure recovery (§8).

6 Proxy Design

The proxy in Obladi has three goals: guarantee good performance, preserve correctness, and guarantee security. To meet these goals, Obladi designs the proxy around the concept of epochs. The proxy partitions time into a set of fixed-length, non-overlapping epochs. Epochs are the granularity at which Obladi guarantees durability and consistency. Each transaction, upon arriving at the proxy, is assigned to an epoch and clients are notified of whether a transaction has committed only when the epoch ends. Until then, Obladi buffers all updates at the proxy.

This flexibility boosts *performance* in two ways. First, it allows Obladi to implement a multiversioned concurrency control (MVCC) algorithm on top of a single versioned Ring ORAM. MVCC algorithms can significantly improve throughput by allowing read operations to proceed with limited blocking. These performance gains are especially significant in the presence of long-running transactions or high storage access latency, as is often the case for cloud storage systems. Second, it reduces traffic to the ORAM, as only the database state at the end of the epoch needs to be written out to cloud storage.

Importantly, Obladi's choice to enforce consistency and durability only at epoch boundaries does not affect *correctness*; transactions continue to observe a serializable and recoverable schedule (i.e., committed transactions do not see writes from aborted transactions).

For transactions executing concurrently within the same epoch, serializability is guaranteed by concurrency control; transactions from different epochs are naturally serialized by the order in which the proxy executes their epochs. No transaction can span multiple epochs; unfinished transactions at epoch boundaries are aborted, so that no transaction is ongoing during epoch changes.

Durability is instead achieved by enforcing epoch fate-sharing [81] during proxy or client crashes: Obladi guarantees that either all *completed* transactions (i.e., transactions for which a commit request has been received) in the epoch are made durable or all transactions abort. This way, no *committed* transaction can ever observe non-durable writes.

Finally, the deterministic pattern of execution that epochs impose drastistically simplifies the task of guaranteeing workload independence: as we describe further below, the frequency and timing at which requests are sent to untrusted storage are fixed and consequently independent of the workload.

The proxy processes epochs with two modules: the concurrency control unit (CCU) ensures that execution remains serializable, while the data handler (DH) accesses the actual data objects. We describe each in turn.

6.1 Concurrency Control

Obladi, like many existing commercial databases [56, 64], uses multiversioned concurrency control [10]. Obladi specifically chooses multiversioned timestamp ordering (MVTSO) [10, 67] because it allows uncommitted writes to be immediately visible to concurrently executing transactions. To ensure serializability, transactions log the set of transactions whose uncommitted values they have observed (their write-read dependencies) and abort if any of their dependencies fail to commit. This optimistic approach is critical to Obladi's performance: it allows transactions within the same epoch to see each other's effects even as Obladi delays commits until the epoch ends. In contrast, a pessimistic protocol like two-phase locking [25], which precludes transactions from observing uncommitted writes, would artificially increase contention by holding exclusive write-locks for the duration of an epoch. When a transaction starts, MVTSO assigns it a unique timestamp that determines its serialization order. A write operation creates a new object version marked with its transaction's timestamp and inserts it in the *version chain* associated with that object. A read operation returns the object's latest version with a timestamp smaller than its transaction's timestamp. Read operations further update a *read marker* on the object's version chain with their transaction's timestamp. Any write operation with

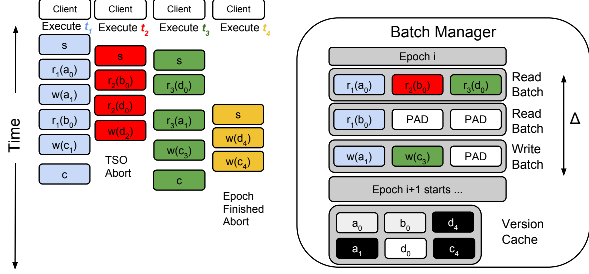


Figure 5: Batching Logic - $r_x(a_y)$ denotes that transaction t_x reads the version of object a written by transaction t_y

a smaller timestamp that subsequently tries to write to this object is aborted, ensuring that no read operation ever fails to observe a write from a transaction that should have preceded it in the serialization order.

Consider for example the set of transactions executing in Figure 5. Transaction t_1's update to object a ($w(a_1)$) is immediately observed by transaction t_3 ($r_3(a_1)$). t_3 becomes dependent on t_1 and can only commit once t_1 also commits. In contrast, t_2's write to object d causes t_2 to abort: a transaction with a higher timestamp (t_3) had already read version d_0, setting the version's read marker to 3.

6.2 Data Handler

Once a version is selected for reading or writing, the DH becomes responsible for accessing or modifying the actual object. Whereas it suffices to guarantee durability and consistency only at epoch boundaries, security must hold at all times, posing two key challenges. First, the number of requests executed in parallel can leak information, e.g., data dependencies within the same transaction [11, 69]. Second, transactions may abort (§6.1), requiring their effects to be rolled back without revealing the existence of contended objects [5, 71]. To decouple the demands of these workloads from the timing and set of requests that it forwards to cloud storage, Obladi leverages the following observation: transactions can always be re-organized so that all reads from cloud storage execute before all writes [19, 37, 46, 87]. Indeed, while operations within a transaction may depend on the data returned by a read from cloud storage, no operation depends on the execution of a write. Accordingly, Obladi organizes the DH into a read phase and a write phase: it first reads all necessary objects from cloud storage, before applying all writes.

Read Phase. Obladi splits each epoch's read phase into a *fixed* set of R *fixed-sized* read batches (b_{read}) that are forwarded to the ORAM executor at *fixed* intervals (Δ_{epoch}). This deterministic structure allows Obladi to execute dependent read operations without revealing the internal control flow of the epoch's transactions. Read operations are assigned to the epoch's next unfilled read batch. If no such batch exists, the transaction is aborted. Conversely, before a batch is forwarded to the ORAM executor, all remaining empty slots are padded with dummy requests. Obladi further *deduplicates* read operations that access the same key. As we describe in §7, this step is necessary for security since parallelized batches may leak information unless requests all access distinct keys [12, 85]. Deduplicating requests also benefits performance by increasing the number of operations that can be served within a fixed-size batch.

Write Phase. While transactions execute, Obladi buffers their write operations into a *version cache* that maintains all object versions created by transactions in the epoch. At the end of an epoch, transactions that have yet to finish executing (recall that epochs terminate at fixed intervals) are aborted and their operations are removed. The latest versions of each object in the version cache according to the version chain are then aggregated in a fixed-size *write batch* (b_{write}) that is forwarded to the ORAM executor, with additional padding if necessary.

This entire process, including write buffering and deduplication, must not violate serializability. The DH guarantees that write buffering respects serializability by directly serving reads from the version cache for objects modified in the current epoch. It guarantees serializability in the presence of duplicate requests by only including the last write of the version chain in a write batch. Since Obladi's epoch-based design guarantees that transactions from a later epoch are serialized after all transactions from an earlier epoch, intermediate object versions can be safely discarded. In this context, MVTSO's requirement that transactions observe the *latest* committed write in the serialization order reduces to transactions reading the tail of the previous epoch's version chain.

In the presence of failures, Obladi guarantees serializability and recoverability by enforcing epoch fate sharing: either all transactions in an epoch are made durable or none are. If a failure arises during epoch e_i, the system simply recovers to epoch e_{i-1}, aborting all transactions in epoch e_i. Once again, this flexibility arises from Obladi delaying commit notifications until epoch boundaries.

Example Execution. We illustrate the batching logic once again with the help of Figure 5. Transactions t_1,t_2,t_3 first execute read operations. These operations are aggregated into the first read batch of epoch i. The values returned by these reads are then *cached* into the version cache. t_2 then executes a write operation, which Obladi also buffers into the version cache. When executing $r_2(d_0)$), t_3 reads object d directly from the version cache (we discuss the security of this step in the next section). Similarly, $r_1(a_1)$ reads the buffered uncommitted version of a. In contrast, Obladi schedules $r_1(b_0)$ to execute as part of the next read batch as b_0 is not present in the version cache. The read batch is then padded to its fixed b_{read} size and executed. t_4 contains no read operations: its write operations are simply executed and buffered at the

version cache. Obladi then finalizes the epoch by aborting all transactions (and their dependencies) that have not yet finished executing: t_4 is consequently aborted. Finally, Obladi aggregates the last version of every update into the write batch (skipping version c_1 of object c for instance, instead only writing c_2), before notifying clients of the commit decision.

6.3 Reducing Work

Obladi reduces work in two additional ways: it caches reads within an epoch and allows Ring ORAM to execute write operations without also executing dummy queries. While these optimizations may appear straightforward, ensuring that they maintain workload independence requires care.

Caching Reads. Ring ORAM maintains a client-side stash (§4) that stores ORAM blocks until their eviction to cloud storage. Importantly, a request for a block present in the stash still triggers a dummy request: a dummy object is still retrieved from each bucket along its path. While this access may appear redundant at first, it is in fact necessary to preserve *workload independence*: removing it removes the guarantee that the set of paths that Obladi requests from cloud storage is uniformly distributed. In particular, blocks present in the stash are more likely to be mapped to paths farther away from the one visited by the last evict path, as they correspond to paths that could not be flushed: buckets have limited space for real blocks and blocks mapped to paths that only intersect near the top of the tree are less likely to find a free slot to which they can be flushed. The degree to which this effect skews the distribution leaks information about the stash size, and, consequently, about the workload. To illustrate, consider the execution in Figure 6. Objects mapped to paths 1 and 2 (a, b, and f) were not flushed from the stash in the previous eviction of path 4. When these objects are subsequently accessed, naively reading them from the stash without performing dummy reads skews the set of paths accessed toward the right subtree (paths 3 and 4)

Obladi securely mitigates some of this work by drawing a novel distinction between objects that are in the stash as a result of a logical access and those present because they could not be evicted. The former can be safely accessed without performing a dummy read, while the latter cannot. Objects present in the stash following a logical access are mapped to independently uniformly distributed paths. Ring ORAM's path invariant ensures that, without caching, the set of accessed paths is uniformly distributed. Removing an independent uniform subset of those paths (namely, the dummy requests) will consequently not change the distribution. Thus, caching these objects, and filling out a read batch with other real or dummy requests, preserves the uniform distribution of paths and leaks no information. Obladi consequently allows all read objects to be placed in the version cache for the duration of the epoch. Objects a, b, d are, for instance, placed in the version cache in Figure 5, allowing read $r_2(d_0)$ to read d directly from the cache. In contrast, objects present in the stash because they could not be evicted are mapped to paths that skew away from the latest evict path. Caching these objects would consequently skew the distribution of requests sent to the storage away from a uniform distribution, as illustrated in Figure 6.

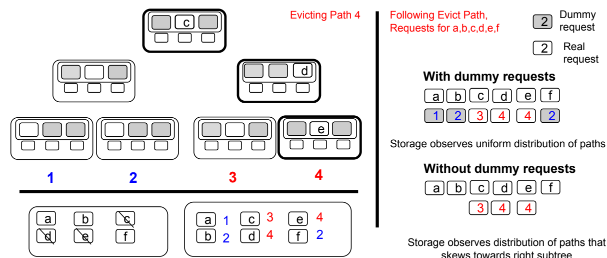


Figure 6: Skew introduced by caching arbitrary objects

Dummiless Writes. Ring ORAM must hide whether requests correspond to read or write operations, as the specific pattern in which these operations are interleaved can leak information [88]; that is why Ring ORAM executes a read operation on the ORAM for every access. In contrast, since transactions can always perform all reads before all writes, no information is leaked by informing the storage server that each epoch consists of a fixed-size sequence of potentially dummy reads followed by a fixed-size sequence of potentially dummy writes. Obladi thus modifies Ring ORAM's algorithm to directly place the new version of an object in the stash, without executing the corresponding read. Note, though, that Obladi continues to increment the evict path count on write operations, a necessary step to preserve the bounds on the stash size, which is important for durability (§8).

6.4 Configuring Obladi

Obladi's good performance hinges on appropriately configuring the size/frequency of batches and ORAM tree for a target application. Table 1 summarizes the parameter space.

Ring ORAM. Configuring Ring ORAM first requires choosing an appropriate Z parameter. Larger values of Z reduce the total size of the ORAM on cloud-storage by decreasing the required height of the ORAM tree and decrease eviction frequency (reducing network/CPU overhead). In contrast, this increase the maximum stash size. Traditional ORAMs thus choose the largest value of Z for which the stash size fits on the proxy. Obladi adds an additional consideration: for durability (as we describe in §8), the stash must be synchronously written out every epoch. One must thus take into account the throughput loss associated with the stash writeback time. Given an appropriate value of Z, Obladi then chooses L, S, and A according to the analytical model proposed in [68].

N	Number of real objects
Z	Number of real slots
S	Number of dummy slots
A	Frequency of evict path
L	Number of levels in the ORAM tree
R	Number of read batches
b_{read}	Size of a read batch
b_{write}	Size of a write batch
Δ	Batch frequency

Table 1: Obladi's configuration parameters

Epochs and batching. Identifiying the appropriate size and number of batches hinges on several considerations. First, Obladi must provision sufficiently many read batches (R) to handle control flow dependencies within transactions. A transaction that executes in sequence five dependent read operations, will for instance require five read batches to execute (it will otherwise repeatedly abort). Second, the ratio of reads ($R * b_{read}$) to writes (w_{write}) must closely approximate the application's read/write ratio. An overly large write batch will waste resources as it will be padded with many dummy requests. A write batch that is too small will lead to frequent aborts caused by the batch filling up. Third, the size of a read or write batch (respectively b_{read} and b_{write}) defines the degree of parallelism that can be extracted. The desired batch size is thus a function of the concurrent load of the system, but also of hardware considerations, as increasing parallelism beyond an I/O or CPU bottleneck serves no purpose. Finally, the number and frequency of read batches within an epoch increases overall latency, but reduces amortized resource costs through caching and operation pipelining (introduced in §7). Latency-sensitive applications may favor smaller batch sizes, while others may prefer longer epochs, but lower overheads.

Security Considerations. Obladi does not attempt to hide the size and frequency of batches from the storage server (we formalize this leakage in §9). Carefully tuning the size and frequency of batches to best match a given application may thus leak information about the application itself. An OLTP application, for instance, will likely have larger batch sizes (b_{read}), but fewer read batches (R), as OLTP applications sustain a high concurrent load of fairly short transactions. OLAP applications will prefer small or non-existent write batches (b_{write}), as they are predominantly read-only, but require many read batches to support the complex joins/aggregates that they implement. Obladi does not attempt to hide the type of application that is being run. It does, however, continue to hide what data is being accessed and what transactions are currently being run at any given point in time. While Obladi's configuration parameters may, for instance, suggest that a medical application like FreeHealth is being run, they do not in any way leak information about how, when, or which patient records are being accessed.

7 Parallelizing the ORAM

Existing ORAM constructions make limited use of parallelism. Some allow requests to execute concurrently between eviction or shuffle phases [12, 69, 85], while others target intra-request parallelism to speed up execution of a single request [43]. Obladi explicitly targets both forms of parallelism. Parallelizing Ring ORAM presents three challenges: (*i*) preserving the correct abstraction of a sequential datastore, (*ii*) enforcing security by concealing the position of real blocks in the ORAM (thereby maintaining workload independence), and (*iii*) preserving existing bounds on the stash size. While these issues also arise in prior work [69], the idiosyncrasies of Ring ORAM add new dimensions to these challenges.

Correctness. Obladi makes two observations. First, while all operations conflict at the Ring ORAM tree's root, they can be split into suboperations that access mostly disjoint *buckets* (§4). Second, conflicting bucket operations can be further parallelized by distinguishing accesses to the bucket's metadata from those to its physical data blocks.

Obladi draws from the theory of multilevel serializability [83], which guarantees that an execution is serializable if the system enforces level-by-level serializability: if operation o is ordered before o' at level i, all suboperations of o must precede conflicting suboperations of o'. Thus, if Obladi orders conflicting operations at a level i, it enforces the same order at level $i+1$ for all their conflicting suboperations; conversely, if two operations do not conflict at level i, Obladi executes their suboperations in parallel. To this end, Obladi simply tracks dependencies across operations and orders conflicting suboperations accordingly. Obladi extracts further parallelism in two ways. First, since in Ring ORAM reads to the same bucket between consecutive eviction or reshuffling operations always target different physical data blocks (even when bucket operations conflict on metadata access), Obladi executes them in parallel. Second, Obladi's own batching logic ensures that requests within a batch touch different objects, preventing read and write methods from ever conflicting. Together, these techniques allow Obladi to execute most requests and evictions in parallel.

We illustrate the dependency tracking logic in Figure 7. The read operation to path 1 conflicts with the evict path for path 2, but only at the root (bucket 1). Thus, reads to buckets 2 and 3 can proceed concurrently, even though accesses to the root's metadata must be serialized, as both operations update the bucket access counter and valid/invalid map (§4).

Security. For security, Obladi's parallel evict path operation must flush the same blocks flushed by a sequential

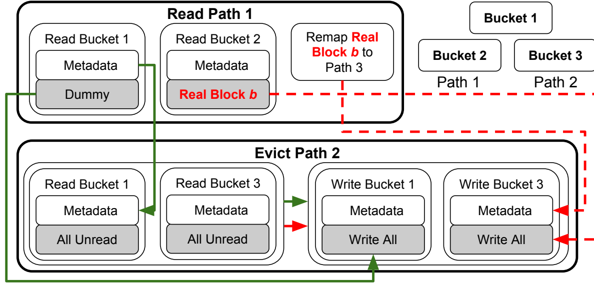


Figure 7: Multilevel Pipelining for a read of path 1 and an evict path of path 2 executing in parallel. Solid green lines represent physical dependencies and dashed red lines represent data dependencies. Inner blocks represent nested operations

implementation. Reproducing this behavior without sacrificing parallelism is challenging. It requires that all real objects brought in during the last A accesses be present in the stash when data is flushed, which may introduce *data dependencies*. Unlike dependencies that arise between operations that access the same physical location in cloud storage, these dependencies are not a deterministic function of an epoch's operations already known to the adversary.

Consider, for instance, block b in Figure 7. In a sequential implementation, b would enter the stash as a result of reading path 1 and be flushed to bucket 3 by the following evict path. Thus, evict path would have to *wait* until b is placed in the stash. Honoring these dependencies opens a timing channel: delay in flushing certain blocks can reveal object placement. As blocks holding real objects can exist anywhere in the tree and be remapped to any path, it follows that it is never secure to execute an eviction operation until all previous access operations have terminated.

Obladi mitigates this restriction by again leveraging delayed visibility and the idea to separate read and write operations within an epoch—but with an important difference. In §6.2 the proxy created separate batches for *logical* read and write operations; to improve parallelism, Obladi, expanding on an idea used by Shroud [43], assigns to separate phases within an epoch the *physical* read and write operations that underlie each of those logical operations. The read phase computes all necessary metadata and executes the set of physical read operations for all logical read path, early reshuffle, and evict path operations. This set is workload independent, so its operations need not be delayed. Physical writes, however, are only flushed at the end of an epoch. The proxy can again apply write deduplication: if a bucket is repeatedly modified during an epoch, only the last version must be written back. Reads that should have read an intermediate write are served locally from the buffered buckets.

The adversary thus always observes a set of reads to random paths followed by a deterministic set of writes independent of the contents of the ORAM and, consequently, of the workload. Data dependencies between read and evict operations no longer create a timing channel. Meanwhile parallelism remains high, as the physical blocks accessed in each phase are guaranteed to be distinct—Ring ORAM directly guarantees this for reads, while bucket deduplication does it for writes.

8 Durability

Obladi guarantees durability at the granularity of epochs: after a crash, it recovers to the state of the last failure-free epoch. Obladi adds two demands to the need of recovering to a consistent state: recovery should leak no information about past or future transactions, and it should be efficient, accessing minimal data from cloud storage. Obladi guarantees the former by ensuring that recovery logic and data logged for recovery maintain workload independence (§3). It strives towards the latter by leveraging the determinism of Ring ORAM.

Consistency. Obladi recovery logic relies on two well-known techniques: write-ahead logging [50] and shadow paging [29]. Obladi mandates that transactions be durable only at the end of an epoch; thus, on a proxy failure, all ongoing transactions can be aborted, and the system reverted to the previous epoch. To make this possible, Obladi must (*i*) recover the proxy metadata lost during the proxy crash, and (*ii*) ensure that the ORAM does not contain any of the aborted transactions' updates. To recover the metadata, Obladi logs three data structures before declaring the epoch committed: the position map, the permutation map, and the stash. The position map and the permutation map identify the position of real objects in the ORAM tree (respectively, in a path and in a bucket); logging them prevents the recovery logic from having to scan the full ORAM to recover the position of buckets. Logging the stash is necessary for correctness. As eviction may be unable to flush the entire stash, some newly written buckets may be present only in the stash, even at epoch boundaries. Failing to log the stash could thus lead to data loss.

To undo partially executed transactions, Obladi adapts the traditional copy-on-write technique of shadow paging [29]: rather than updating buckets in place, it creates new versions of each bucket on every write. Obladi then leverages the inherent determinism of Ring ORAM to reconstruct a consistent snapshot of the ORAM at a given epoch. In Ring ORAM, the current version of a bucket (i.e. the number of times a bucket has been written) is a deterministic function of the number of prior evict paths. The number of evict paths per epoch is similarly fixed (evict paths happen every A accesses, and epochs are of fixed size). Obladi can then trivially revert the ORAM on failures by setting the evict path counter to its value at the end of the last committed epoch. This counter determines the number of evict paths that have occurred, and

consequently the object versions of the corresponding epoch.

Security. Obladi ensures that (*i*) the information logged for durability remains independent of data accesses, and (*ii*) that the interactions between the failed epoch, the recovery logic, and the next epoch preserve workload independence.

Obladi addresses the first issue by encrypting the position map and the contents of the permutations table. It similarly encrypts the stash, but also *pads* it to its maximum size, as determined in canonical Ring ORAM [68], to prevent it from indicating skew (if a small number of objects are accessed frequently, the stash will tend to be smaller).

The second concern requires more care: workload independence must hold before, during, and after failures. Ring ORAM guarantees workload independence through two invariants: the bucket invariant and the path invariant (§4). Preserving bucket slots from being read twice between evictions is straightforward. Obladi simply logs the invalid/valid map to track which slots have already been read and recovers it during recovery; there is no need for encryption, as the set of slots read is public information. Ensuring that the ORAM continues to observe a uniformly distributed set of paths is instead more challenging. Specifically, read requests from partially executed transactions can potentially leak information, even when recovering to the previous epoch. Traditionally, databases simply *undo* partially executed transactions, mark them as aborted, and proceed as if they had never existed. From a security standpoint, however, these transactions were still observed by the adversary, and thus may leak information. Consider a transaction accessing object *a* (mapped to path 1) that aborts because of a proxy failure. Upon recovery, it is likely that a client will attempt to access *a* again. As the recovery logic restores the position map of the previous epoch, that new operation on *a* will result in another access to path 1, revealing that the initial access to path 1 was likely real (rather than padded), as the probability of collisions between two uniformly chosen paths is low. To mitigate this concern while allowing clients to request the same objects after failure, Obladi durably logs the list of paths and slot indices that it accesses, before executing the actual requests, and replays those paths during recovery (remapping any real blocks). While this process is similar to traditional database redo logging [50], the goal is different. Obladi does not try to reapply transactions (they have all aborted), but instead forces the recovery logic to be deterministic: the adversary always sees the paths from the aborted epoch repeated after a failure.

Optimizations. To minimize the overhead of checkpointing, Obladi checkpoints deltas of the position, permutation, and valid/invalid map, and only periodically checkpoints the full data structures. While the number of changes to the permutation and valid/invalid maps directly follows from the set of physical requests made to cloud storage, the size of the delta for the position map reveals how many real requests were included in an epoch—padded requests do not lead to position map updates. Obladi thus pads the map delta to the maximum number of entries that could have changed in an epoch (i.e., the read batch size times the number of read batches, plus the size of the single write batch).

9 System Security

We now outline Obladi's security guarantees, deferring a formal treatment to Appendix B. To the best of our knowledge, we are the first to formalize the notion of crashes in the context of oblivious RAM.

Model We express our security proof within the Universal Composability (UC) framework [14], as it aligns well with the needs of modern distributed systems: a UC-secure system remains UC-secure under concurrency or if composed with other UC-secure systems. Intuitively, proving security in the UC model proceeds as follows. First, we specify an *ideal functionality* $\mathcal{F}$ that defines the expected functionality of the protocol for both correctness and security. For instance, Obladi requires that the execution be serializable, and that only the frequency of read and write batches be learned. We must ensure that the real protocol provides the same functionality to honest parties while leaking no more information than $\mathcal{F}$ would. To establish this, we consider two different worlds: one where the real protocol interacts with an adversary $\mathcal{A}$, and one where $\mathcal{F}$ interacts with $\mathcal{S}_{\mathcal{A}}$, our best attempt at simulating $\mathcal{A}$. $\mathcal{A}$'s transcript—including its inputs, outputs, and randomness—and $\mathcal{S}_{\mathcal{A}}$'s output are given to an environment $\mathcal{E}$, which can also observe all communications within each world. $\mathcal{E}$'s goal is to determine which world contains the real protocol. To prompt the worlds to diverge, $\mathcal{E}$ can delay and reorder messages, and even control external inputs (potentially causing failures). Intuitively, $\mathcal{E}$ represents anything external to the protocol, such as concurrently executing systems. We say that the real protocol is secure if, for any adversary $\mathcal{A}$, we can construct $\mathcal{S}_{\mathcal{A}}$ such that $\mathcal{E}$ can never distinguish between the worlds.

Assumptions The security of Obladi relies on four assumptions. (*i*) Canonical Ring ORAM is linearizable (*ii*) MVTSO generates serializable executions. (*iii*) The network will retransmit dropped packets. The adversary learns of the retransmissions, but nothing more.

Ideal Functionality To define the ideal functionality $\mathcal{F}_{Ob}$, recall that the proxy is considered trusted while interactions with the cloud storage are not. This allows $\mathcal{F}_{Ob}$ to replace the proxy and intermediate between clients and the storage server, performing the same functions as the proxy (we do not try to hide the concurrency/batching

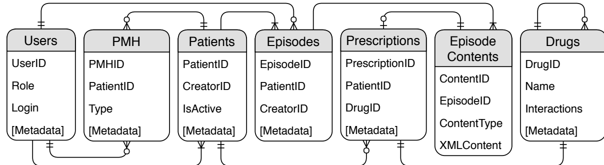


Figure 8: FreeHealth Database Architecture

logic). We must, however, define $\mathcal{F}_{Ob}$ to obliviously hide data values and access patterns. To this end, when the proxy logic finalizes a batch, $\mathcal{F}_{Ob}$ simply informs the storage server that it is executing a read or write batch. Since $\mathcal{F}_{Ob}$ is a theoretical ideal, we allow it to manage all storage internally, so it then updates its local storage and furnishes the appropriate response to each client.

In this setup, modeling proxy crashes is straightforward. Crashes can occur at any time and cause the proxy to lose all state. So, on an external input to crash, $\mathcal{F}_{Ob}$ simply clears its state. Since we accept that $\mathcal{A}$ may learn of proxy crashes, $\mathcal{F}_{Ob}$ also sends a message to the storage server that it has crashed.

Proof Sketch The correctness of the system is straightforward, as $\mathcal{F}_{Ob}$ behaves much the same as the proxy.

To prove security, we must demonstrate that, for any algorithm $\mathcal{A}$ defining the behavior of the storage server, we can accurately simulate $\mathcal{A}$'s behavior using only the information provided by $\mathcal{F}_{Ob}$. Note that the simulator $\mathcal{S}_{\mathcal{A}}$ can run $\mathcal{A}$ internally, as $\mathcal{A}$ is simply an algorithm. Thus we can define $\mathcal{S}_{\mathcal{A}}$ to operate as follows. When $\mathcal{S}_{\mathcal{A}}$ receives notification of a batch, it constructs a parallel ORAM batch from uniformly random accesses of the correct type. It provides these accesses to $\mathcal{A}$ and produces $\mathcal{A}$'s response.

The security of this simulation hinges on two key properties: (*i*) the caching and deduplication logic do not affect the distribution of physical accesses, and (*ii*) the physical access pattern of a parallelized batch is entirely determined by the physical accesses proscribed by sequential Ring ORAM for the same batch. The first follows from Ring ORAM's guarantee that each access will be an independent uniformly random path—removing an independently-sampled element does not change the distribution of the remaining set. The second follows from the parallelization procedure simply aggregating all accesses and performing all reads followed by all writes.

These properties ensure that the random access pattern produced by $\mathcal{S}_{\mathcal{A}}$ is identical to the access pattern produced by the proxy when operating on real data. Thus the simulated $\mathcal{A}$ must behave exactly as it would when provided with real data, and produce indistinguishable output.

10 Implementation

Our prototype consists of 41,000 lines of Java code. We use the Netty library for network communication (v4.1.20), Google protobuffers for serialization (v3.5.1), the Bouncy Castle library (v1.59) for encryption, and the Java MapDB library (v3) for persistence. We additionally implement a non-private baseline (NoPriv). NoPriv shares the same concurrency control logic (TSO), but replaces the proxy data handler with non-private remote storage. NoPriv neither batches nor delays operations; it buffers writes at the local proxy until commit, and serves writes locally when possible.

11 Evaluation

Obladi leverages the flexibility of transactional commits to mitigate the overheads of ORAM. To quantify the benefits and limitations of this approach, we ask:

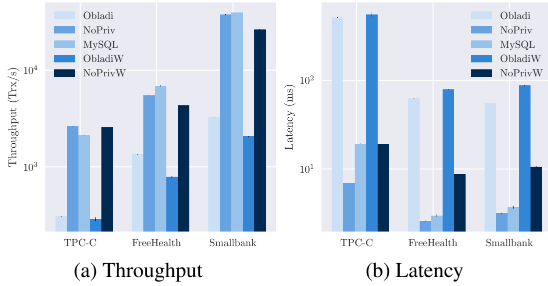
1. How much does Obladi pay for privacy? (§11.1)
2. How do epochs affect these overheads? (§11.2)
3. Can Obladi recover efficiently from failures? (§11.3)

Experimental Setup The proxy runs on a c5.xlarge Amazon EC2 instance (16 vCPUs, 32GB RAM), and the storage on an m5.4xlarge instance (16 vCPUs, 64GB RAM). The ORAM tree is configured with $Z = 100$ and optimal values of S and A (respectively, 196 and 168) [68]. We report the average of three 90 seconds runs (30 seconds ramp-up/down).

Benchmarks We evaluate the performance of our system using three applications: TPC-C [21, 79], SmallBank [21], and FreeHealth [27, 41]. Our microbenchmarks use the YCSB [18] workload generator. **TPC-C**, the defacto standard for OLTP workloads, simulates the business logic of e-commerce suppliers. We configure TPC-C to run with 10 warehouses [86]. In line with prior transactional key-value stores [78], we use a separate table as a secondary index on the `order` table to locate a customer's latest order in the `order status` transaction, and on the `customer` table to look up customers by their last names (`order status` and `payment`). **Smallbank** [21] models a simple banking application supporting money transfers, withdrawals, and deposits. We configure it to run with one million accounts. Finally, we port **FreeHealth** [27, 41], an actively-used cloud EHR system (Figure 8). FreeHealth supports the business logic of medical practices and hospitals. It consists of 21 transaction types that doctors use to create patients and look up medical history, prescriptions, and drug interactions.

11.1 End-to-end Performance

Figure 9 summarizes the results from running the three end-to-end applications in two setups: a local setup in which the latency between proxy and server is low (0.3ms) (**Obladi, NoPriv**), and a more realistic WAN setup with 10ms latency (**ObladiW**, **NoPrivW**). We additionally compare those results with a local MySQL setup. MySQL, unlike NoPriv, cannot buffer writes. We consequently do not evaluate MySQL in the WAN setting.

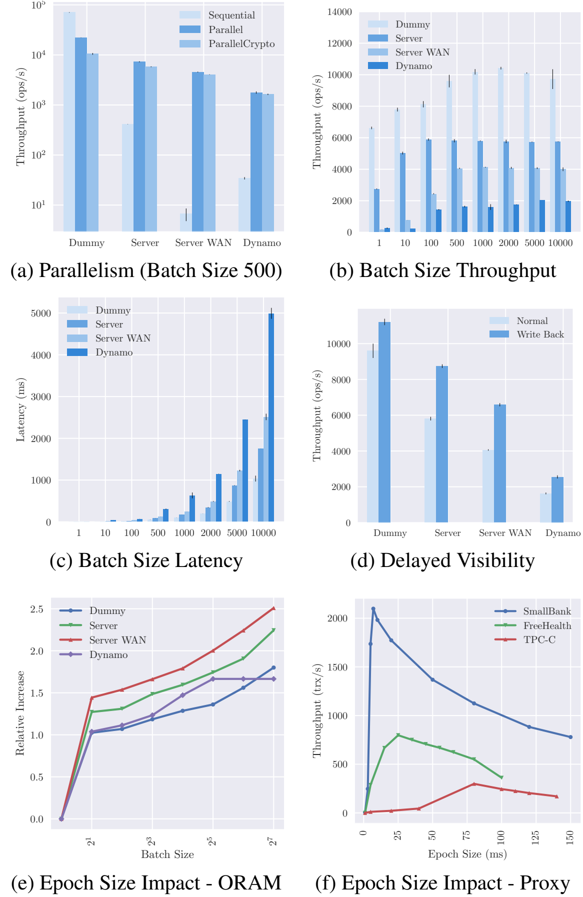


(a) Throughput (b) Latency

Figure 9: Application Performance

TPC-C Obladi comes within 8× of NoPriv's throughput, as NoPriv is contention-bottlenecked on the high rate of conflicts between the `new-order` and `payment` transactions on the `district` table. NoPriv's performance is itself slightly higher than MySQL as the use of MVTSO allows for the `new-order` and `payment` transactions to be pipelined. In contrast, MySQL acquires exclusive locks for the duration of the transactions. Latency, however, spikes to 70× over NoPriv because of the inflexible execution pattern Obladi needs for security. Transactions in TPC-C vary heavily in size. Epochs must be large enough to accommodate all transactions, and hence artificially increase the latency of short instances. Moreover, write operations must be applied atomically during epoch changes. For a write batch size of 2,000, this process takes on average 340ms, further increasing latency for individual transactions. The write-back process also limits throughput, even preventing non-conflicting operations from making progress (in contrast, NoPriv can benefit from writes never blocking reads in MVTSO). Epoch changes also introduce additional aborts for transactions that straddle epochs. The additional 10ms latency of the WAN setting has comparatively little effect, as the large write batch size of TPC-C is the primary bottleneck: throughput remains within 9x of NoPrivW. Also NoPrivW's performance does not degrade: since MVTSO exposes uncommitted writes immediately, increasing commit latency does not increase contention.

Smallbank Transactions in Smallbank are more homogeneous (between three and six operations); thus, the length of an epoch can be set to more closely approximate most transactions, reducing latency overheads (17× NoPriv). NoPriv is CPU bottlenecked for Smallbank; the relative throughput drop for Obladi is higher (12×) because of the overhead of changing epochs and the blocking that it introduces. Transaction dependency tracking becomes a bottleneck in NoPriv, resulting in a 15% throughput loss over MySQL. Increasing latency between proxy and storage causes both systems' throughput to drop. ObladiW's 35% drop is due to the increased duration of epoch changes (during which no other transactions can execute) while NoPrivW's 30% drop stems from the larger dependency chains that arise from the relatively long commit phase.



(a) Parallelism (Batch Size 500) (b) Batch Size Throughput

(c) Batch Size Latency (d) Delayed Visibility

(e) Epoch Size Impact - ORAM (f) Epoch Size Impact - Proxy

Figure 10: Performance impact of various features

FreeHealth Like SmallBank, FreeHealth consists of fairly short transactions and can thus choose a fairly small epoch (five read batches), reducing the impact on latency (20× NoPriv). Unlike Smallbank, however, FreeHealth consists primarily of read operations, and so it can choose a much smaller write batch (200), minimizing the cost of epoch changes and maximizing throughput (only a 4× drop over NoPriv and a 5.5× over NoPrivW for ObladiW). Both NoPriv and Obladi are contention-bottlenecked on the creation of *episodes*, the core units of EHR systems that encapsulate prescriptions, medical history, and patient interaction.

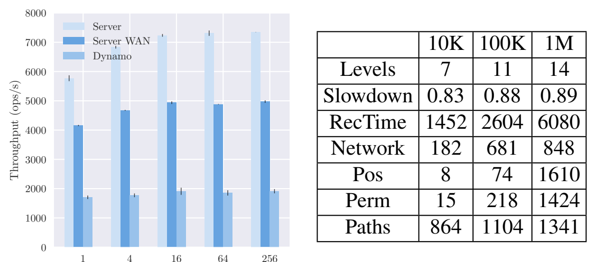
11.2 Impact of Epochs

Though epochs create blocking and cause aborts, they are key to reducing the cost of accessing ORAM, as they allow to (*i*) securely parallelize the ORAM and (*ii*) delay and buffer bucket writes. To quantify epochs' impact on performance as a function of their size and the underlying storage properties, we instantiate an ORAM with 100K objects and choose three different storage backends: a local dummy (storing no real data) that responds to all reads with a static value and ignores writes (`dummy`); a remote

server backend with an in-memory hashmap (`server`, ping time 0.3ms) and a remote WAN server backend with an in-memory hashmap (`server WAN`, ping time 10ms); and DynamoDB (`dynamo`, provisioned for 80K req/s, read ping 1ms, write 3ms).

Parallelization We first focus on the performance impact of parallelizing Ring ORAM (ignoring other optimizations). Graph 10a shows that, unsurprisingly, the benefits of parallelism increase with the latency of individual requests. Parallelizing the ORAM for `dummy`, for instance, yields no performance gain; in fact, it results in a 3× slowdown (from 72K req/s to 24K req/s). Sequential Ring ORAM on `dummy` is CPU-bound on metadata computation (remapping paths, shuffling buckets, etc.), so adding coordination mechanisms to guarantee multi-level serializability only increases the cost of accessing a bucket. As storage access latency increases and the ORAM becomes I/O-bound, the benefits of parallelism become more salient. For a batch size of 500, throughput increases by 12× for `server`, as much as 51× for `dynamo`, and 510×for `WAN server`. The available parallelism is a function of both the size/fan-out of the tree and the underlying resource bottlenecks of the proxy. Graph 10b captures the parallelization speedup for both intra- and inter-request parallelism, while Graph 10b quantifies the latency impact of batching. The parallelization speedup achieved for a batch size of one captures intra-request parallelism: the eleven levels of the ORAM can be accessed concurrently, yielding an 11× speedup. As batch sizes increase, Obladi can leverage inter-request parallelism to process non-conflicting physical operations in parallel, with little to no impact on latency. `Dynamo` peaks early (at 1750 req/s) because its client API uses blocking HTTP calls, and `dummy`'s storage eventually bottlenecks on encryption, but `server` and `WAN server` are more interesting. Their throughput is limited by the physical and data dependencies on the upper levels of the tree (recall that paths always conflict at the root (§7)).

Work Reduction To amortize ORAM overheads across a large number of operations, Obladi relies on delayed visibility to buffer bucket writes until the end of an epoch, when they can be executed in parallel, discarding intermediate writes. Reads to those buckets are directly served from the proxy, reducing network communication and CPU work (as encryption is not needed). Graph 10d shows that enabling this optimization for an epoch of eight batches (a setup suitable for FreeHealth and TPC-C) yields a 1.5× speedup on both `dynamo` and the server, a 1.6× speedup on the WAN server, but only minimal gains for `dummy` (1.1×). When using a small number of batches, throughput gains come primarily from combining duplicate operations in buckets near the top of the tree. For example, the root bucket is written 27 times in an epoch of size eight (once per eviction, every 168 requests). As these



	10K	100K	1M
Levels	7	11	14
Slowdown	0.83	0.88	0.89
RecTime	1452	2604	6080
Network	182	681	848
Pos	8	74	1610
Perm	15	218	1424
Paths	864	1104	1341

(a) Checkpoint Frequency (100K) (b) Server Wan Recovery Time (ms)

Figure 11: Durability

operations conflict, they must be executed sequentially and quickly become the bottleneck (other buckets have fewer operations to execute). Our optimization lets Obladi write the root bucket only once, significantly reducing latency and thus increasing throughput. As epochs grow in size, increasingly many buckets are buffered locally until the end of the epoch (§7), allowing reads to be served locally and further reducing I/O with the storage. Consider Graph 10e: throughput increases almost logarithmically; metadata computation eventually becomes a bottleneck for `dummy`, while `server` and `server WAN` eventually run out of memory from storing most of the tree (our AWS account did not allow us to provision `dynamo` adequately for larger batches). Larger epochs reduce the raw amount of work per operation: with one batch, Obladi requires 41 physical requests per logical operation, but only requires 24 operations with eight batches. For real transactional workloads, however, epochs are not a silver bullet. Graph 10f suggests that applications are very sensitive to identifying the right epoch duration: too short and transactions cannot make progress, repeatedly aborting; too long and the system will remain unnecessarily idle.

11.3 Durability

Table 11b quantifies the efficiency of failure recovery and the cost it imposes on normal execution for ORAMS of different sizes (we show space results for only the WAN server as Dynamo follows a similar trend). During normal execution, durability imposes a moderate throughput drop (from 0.83× for 10K to 0.89× for 1M). This slowdown is due to the need to checkpoint client metadata and to synchronously log read paths to durable storage before reading. As seen in Graph 11a, computing diffs mitigates the impact of checkpointing. Recovery time similarly increases as the ORAM grows, from 1.5s to 6.1s (Table 11b, *RecTime*). The costs of decrypting the position and permutation maps (*Pos* and *Perm*) are low for small datasets, but grow linearly with the number of keys. Read path logging (*Paths*) instead starts much larger, but grows only with the depth of the tree.

12 Related Work

Batching Obladi amortizes ORAM costs by grouping operations into epochs and committing at epoch boundaries. Batching can mitigate expensive security primitives, e.g., it reduces server-side computation in private information retrieval (PIR) schemes [9, 30, 32, 44], amortizes the cost of shuffling networks in Atom [39] and the cost of verifying integrity in Concerto [6]. Changing when operations output commit is a popular performance-boosting technique: it yields significant gains for state-machine replication [34, 36, 63], file systems [54], and transactional databases [20, 46, 81].

ORAM parallelism Obladi extends recent work on parallel ORAM constructions [11, 43, 85] to extract parallelism both *within* and *across* requests. Shroud [43] targets intra-request parallelism by concurrently accessing different levels of tree-based ORAMs. Chung et al [12] and PrivateFS [85] instead target inter-request parallelism, respectively in tree-based [72] and hierarchical [84] ORAMs. Both works execute requests to distinct logical keys concurrently between reshuffles or evictions and deduplicate concurrent requests for the same key to increase parallelism. Obladi leverages delayed visibility to separate batches into read and write phases, extracting concurrency both within requests and across evictions. Furthermore, Obladi parallelizes across requests by deduplicating requests at the trusted proxy.

ObliviStore [76] and Taostore [69] instead approach parallelization by focusing on asynchrony. ObliviStore [76] formalizes the security challenges of scheduling requests asynchronously; the oblivious scheduling mechanism that it presents for that model however is computationally expensive and requires a large stash, making ObliviStore unsuitable for implementing ACID transactions. Like ObliviStore, Taostore leverages asynchrony to parallelize Path ORAM [77], a tree-based construction from which Ring ORAM descends. Taostore, however, targets a different threat model: it assumes both that requests must be processed immediately, and that the timing of responses is visible to the adversary. Request latencies thus necessarily increase linearly with the number of clients [85].

Hiding access patterns for non-transactional systems Many systems seek to provide access pattern protections for analytical queries: Opaque [88] and Cipherbase [5] support oblivious operators for queries that scan or shuffle full tables. Both rely on hardware enclaves for efficiency: Opaque runs a query optimizer in SGX [31], while Cipherbase leverages secure co-processors to evaluate predicates more efficiently. Others seek to hide the parameters of the query rather than the query itself: Olumofin et al. [55] do it via multiple rounds of keyword-based PIR operations [16]; Splinter [82] reduces the number of round-trips necessary by mapping these database queries to function secret sharing primitives. Finally, ObliDB [24] adds support for point queries and efficient updates by designing an oblivious B-tree for indexing. The concurrency control and recovery mechanisms of all these approaches introduce timing channels and structure writes in ways that leak access patterns [5].

Encryption Many commercial systems offer the possibility to store encrypted data [23, 70]. Efficiently executing data-dependent queries like joins, filters, or aggregations without knowledge of the plaintext is challenging: systems like CryptDB [62], Monomi [80], and Seabed [59] tailor encryption schemes to allow executing certain queries directly on encrypted data. Others leverage trusted hardware [7]. In contrast, executing transactions on encrypted data is straightforward: neither concurrency control nor recovery requires knowledge of the plaintext data.

13 Conclusion

This paper presents Obladi, a system that, for the first time, considers the security challenges of providing ACID transactions without revealing access patterns. Obladi guarantees security and durability at moderate cost through a simple observation: transactional guarantees are only required to hold for committed transactions. By delaying commits until the end of epochs, Obladi inches closer to providing practical oblivious ACID transactions.

Acknowledgements We thank our shepherd, Jay Lorch, for his commitment to excellence, and the anonymous reviewers for their helpful comments. We are grateful to Sebastian Angel, Soumya Basu, Vijay Chidambaram, Trinabh Gupta, Paul Grubbs, Malte Schwarzkopf, Yunhao Zhang, and the MIT PDOS reading group for their feedback. This work was supported by NSF grants CSR-1409555 and CNS-1704742, and an AWS EC2 Education Research grant.

References

[1] AGRAWAL, D., AND EL ABBADI, A. Locks with Constrained Sharing (Extended Abstract).

[2] AGUILAR-MELCHOR, C., BARRIER, J., FOUSSE, L., AND KILLIJIAN, M.-O. XPIR: Private Information Retrieval for Everyone. Cryptology ePrint Archive, Report 2014/1025, 2014. http://eprint.iacr.org/2014/1025.

[3] AMAZON. S3: Simple storage service. https://aws.amazon.com/s3/.

[4] AMAZON. Simple db. https://aws.amazon.com/simpledb/.

[5] ARASU, A., BLANAS, S., EGURO, K., KAUSHIK, R., KOSSMANN, D., RAMAMURTHY, R., AND VENKATESAN, R. Orthogonal Security With Cipherbase. In *Conference on Innovative Data Systems Research (CIDR)* (2013).

[6] ARASU, A., EGURO, K., KAUSHIK, R., KOSSMANN, D., MENG, P., PANDEY, V., AND RAMAMURTHY, R. Concerto: A High Concurrency Key-Value Store with Integrity. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2017).

[7] BAJAJ, S., AND SION, R. TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2011).

[8] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)* (2011).

[9] BEIMEL, A., ISHAI, Y., AND MALKIN, T. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology (JOFC) 17*, 2 (2004), 125–151.

[10] BERNSTEIN, P. A., AND GOODMAN, N. Multiversion Concurrency Control — Theory and Algorithms. *ACM Trans. Database Syst. 8*, 4 (1983), 465–483.

[11] BINDSCHAEDLER, V., NAVEED, M., PAN, X., WANG, X., AND HUANG, Y. Practicing Oblivious Access on Cloud Storage: The Gap, the Fallacy, and the New Way Forward. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[12] BOYLE, E., CHUNG, K.-M., AND PASS, R. Oblivious Parallel RAM and Applications. In *Theory of Cryptography Conference (TCC)* (2016).

[13] BULCK, J. V., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium (USENIX)* (2018).

[14] CANETTI, R. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science (FOCS)* (2001).

[15] CECCHETTI, E., ZHANG, F., JI, Y., KOSBA, A., JUELS, A., AND SHI, E. Solidus: Confidential Distributed Ledger Transactions via PVORM. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[16] CHOR, B., GILBOA, N., AND NAOR, M. Private information retrieval by keywords, 1997.

[17] CLOUD, C. 5 advantages of a cloud-based EHR.

[18] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)* (2010).

[19] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8:1–8:22.

[20] CROOKS, N., PU, Y., ALVISI, L., AND CLEMENT, A. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *ACM Symposium on Principles of Distributed Computing (PODC)* (2017).

[21] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRE-MAUROUX, P. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases.

[22] DYNAMODB. DynamoDB. https://aws.amazon.com/dynamodb/.

[23] DYNAMODB. Encryption at rest. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/EncryptionAtRest.html.

[24] ESKANDARIAN, S., AND ZAHARIA, M. An Oblivious General-Purpose SQL Database for the Cloud. *CoRR abs/1710.00458* (2017).

[25] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM 19*, 11 (1976), 624–633.

[26] FLETCHER, C. W., REN, L., KWON, A., AND V. DI, M. A Low-Latency, Low-Area Hardware Oblivious RAM Controller. In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2015).

[27] FREEHEALTH.IO. FreeHealth EHR. https://freehealth.io/. Accessed 2018-05-01.

[28] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[29] GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys (CSUR) 13*, 2 (1981), 223–242.

[30] GUPTA, T., CROOKS, N., MULHERN, W., SETTY, S., ALVISI, L., AND WALFISH, M. Scalable and Private Media Consumption with Popcorn. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[31] INTEL. Intel Software Guard Extension - SGX. https://software.intel.com/en-us/sgx.

[32] ISHAI, Y., KUSHILEVITZ, E., OSTROVSKY, R., AND SAHAI, A. Batch Codes and Their Applications. In *ACM Symposium on Theory of Computing (STOC)* (2004).

[33] JONES, E. P., ABADI, D. J., AND MADDEN, S. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2010).

[34] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., AND DAHLIN, M. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012).

[35] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (SP)* (2019).

[36] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems (TOCS) 27*, 4 (2010), 7:1–7:39.

[37] KUNG, H. T., AND ROBINSON, J. T. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst. 6*, 2 (1981), 213–226.

[38] KUO, A. M.-H. Opportunities and challenges of cloud computing to improve health care services. *Journal of Medical Internet Research (JMIR) 13*, 3 (2011).

[39] KWON, A., CORRIGAN-GIBBS, H., DEVADAS, S., AND FORD, B. Atom: Horizontally Scaling Strong Anonymity. In *ACM Symposium on Operating System Principles (SOSP)* (2017).

[40] LARSON, P.-A., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-performance Concurrency Control Mechanisms for Main-memory Databases. In *Proceedings of the VLDB Endowment (PVLDB)* (2011).

[41] LIBRE, M. FreeHealth EHR. https://https://freemedsoft.com/fr/. Accessed 2018-05-01.

[42] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium (USENIX)* (2018).

[43] LORCH, J., PARNO, B., MICKENS, J., RAYKOVA, M., AND SCHIFFMAN, J. Shroud: Ensuring Private Access to Large-Scale Data in the Data Center. In *Conference on File and Storage Technologies (FAST)* (2013).

[44] LUEKS, W., AND GOLDBERG, I. Sublinear Scaling for Multi-Client Private Information Retrieval. In *Financial Cryptography and Data Security (FC)* (2015).

[45] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[46] MEHDI, S. A., LITTLEY, C., CROOKS, N., ALVISI, L., BRONSON, N., AND LLOYD, W. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[47] MICROSOFT. Azure tables. https://azure.microsoft.com/en-us/services/storage/tables/.

[48] MICROSOFT. Documentdb - nosql service for json. https://azure.microsoft.com/en-us/services/documentdb/.

[49] MICROSOFT. SQL Server. https://www.microsoft.com/en-cy/sql-server/sql-server-2016.

[50] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst. 17*, 1 (1992), 94–162.

[51] MONGODB. Agility, Performance, Scalibility. Pick three. https://www.mongodb.org/.

[52] NARAYANAN, A., AND SHMATIKOV, V. Robust De-anonymization of Large Sparse Datasets. In *IEEE Symposium on Security and Privacy (SP)* (2008).

[53] NARAYANAN, A., AND SHMATIKOV, V. Myths and fallacies of "personally identifiable information". *Commun. ACM 53*, 6 (June 2010), 24–26.

[54] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the Sync. *ACM Transactions on Computer Systems (TOCS) 26*, 3 (2008), 6:1–6:26.

[55] OLUMOFIN, F., AND GOLDBERG, I. Privacy-preserving Queries over Relational Databases. In *Privacy Enhancing Technologies Symposium (PETS)* (2010).

[56] ORACLE. InnoDB. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html/.

[57] ORACLE. MySQL. https://www.mysql.com/.

[58] ORACLE. MySQL Cluster. https://www.mysql.com/products/cluster/.

[59] PAPADIMITRIOU, A., BHAGWAN, R., CHANDRAN, N., RAMJEE, R., HAEBERLEN, A., SINGH, H., MODI, A., AND BADRINARAYANAN, S. Big Data Analytics over Encrypted Datasets with Seabed. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[60] PAPADIMITRIOU, C. H. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM) 26*, 4 (1979), 631–653.

[61] PLATFORM, G. C. Cloud spanner. http://cloud.google.com/spanner/.

[62] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM Symposium on Operating System Principles (SOSP)* (2011).

[63] PORTS, D. R., LI, J., LIU, V., SHARMA, N. K., AND KRISHNAMURTHY, A. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[64] POSTGRESQL. http://www.postgresql.org/.

[65] REDDY, P. K., AND KITSUREGAWA, M. Speculative Locking Protocols to Improve Performance for Distributed Database Systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 16*, 2 (2004), 154–169.

[66] REED, D. P. Implementing Atomic Actions on Decentralized Data (Extended Abstract). In *ACM Symposium on Operating System Principles (SOSP)* (1979).

[67] REED, D. P. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems (TOCS) 1*, 1 (1983), 3–23.

[68] REN, L., FLETCHER, C., KWON, A., STEFANOV, E., SHI, E., VAN DIJK, M., AND DEVADAS, S. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security Symposium (USENIX)* (2015).

[69] SAHIN, C., ZAKHARY, V., EL ABBADI, A., LIN, H., AND TESSARO, S. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *IEEE Symposium on Security and Privacy (SP)* (2016).

[70] SERVER, M. S. Always Encrypted. https://www.microsoft.com/en-us/research/project/always-encrypted/.

[71] SHEFF, I., MAGRINO, T., LIU, J., MYERS, A. C., AND VAN RENESSE, R. Safe Serializable Secure Scheduling: Transactions and the Trade-Off Between Security and Consistency. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[72] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *International Conference on The Theory and Application of Cryptology and Information Security* (2011).

[73] SINGEL, R. Netflix spilled your *Brokeback Mountain* secret, lawsuit claims. *Wired* (Dec. 2009). http://www.wired.com/images_blogs/threatlevel/2009/12/doe-v-netflix.pdf.

[74] STEFANOV, E., AND SHI, E. ObliviStore: High Performance Oblivious Cloud Storage. In *IEEE Symposium on Security and Privacy (SP)* (2013).

[75] STEFANOV, E., AND SHI, E. ObliviStore: High Performance Oblivious Distributed Cloud Data Store. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[76] STEFANOV, E., SHI, E., AND SONG, D. Towards Practical Oblivious RAM.

[77] 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 *ACM Conference on Computer and Communications Security (CCS)* (2013).

[78] SU, C., CROOKS, N., DING, C., ALVISI, L., AND XIE, C. Bringing Modular Concurrency Control to the Next Level. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2017).

[79] TRANSACTION PROCESSING PERFORMANCE COUNCIL. The TPC-C home page. http://www.tpc.org/tpcc.

[80] TU, S., KAASHOEK, M. F., MADDEN, S., AND ZELDOVICH, N. Processing Analytical Queries over Encrypted Data. In *Proceedings of the VLDB Endowment (PVLDB)* (2013).

[81] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy Transactions in Multicore In-memory Databases. In *ACM Symposium on Operating System Principles (SOSP)* (2013).

[82] WANG, F., YUN, C., GOLDWASSER, S., VAIKUNTANATHAN, V., AND ZAHARIA, M. Splinter: Practical Private Queries on Public Data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[83] WEIKUM, G. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Trans. Database Syst. 16*, 1 (1991), 132–180.

[84] WILLIAMS, P., SION, R., AND CARBUNAR, B. Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage. In *ACM Conference on Computer and Communications Security (CCS)* (2008).

[85] WILLIAMS, P., SION, R., AND TOMESCU, A. PrivateFS: A Parallel Oblivious File System. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[86] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance ACID via Modular Concurrency Control. In *ACM Symposium on Operating System Principles (SOSP)* (2015).

[87] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building Consistent Transactions with Inconsistent Replication. In *ACM Symposium on Operating System Principles (SOSP)* (2015).

[88] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

A Ensuring Data Integrity in Obladi

As we described in §3, we assume the untrusted storage server is *honest-but-curious*. In many cases this is a very strong assumption that system operators may not be happy to make. We can remove this requirement with the use of Message Authentication Codes (MACs) and a trusted counter—used to ensure freshness—that persists across crashes. We describe this technique here.

When we assumed the server was honest-but-curious, we assumed it could deny service, but would otherwise correctly respond to all queries. In order to remove this assumption while maintaining security, we must create a means to identify if the storage server returns incorrect data, thus reducing them to DoS attacks. To do this, the proxy must verify that the returned value is the value (*i*) most recently written (*ii*) by the proxy (*iii*) to the specified location.

We can guarantee (*ii*) using MACs. At initialization, the proxy generates a secret MAC key (in addition to its secret encryption key) and attaches a MAC to every piece of data it stores on the cloud server. This allows the proxy to verify that the cloud server did not modify the data or manufacture its own.

By themselves, MACs do not guarantee (*i*) and (*iii*), as the cloud server can provide an old copy of the data or valid data from a different location, both of which will have valid MACs. We additionally need to include a unique identifier that the proxy can easily recompute. For data that is written at most once per epoch, this unique identifier can be the pair of epoch, ORAM location. Due to Ring ORAM's deterministic eviction algorithm, the proxy can compute the epoch during which any given block was most recently written knowing only the current epoch counter and the early reshuffle table.

There is exactly one value which is written multiple times per epoch: each read batch, of which there may be many per epoch, logs the accessed locations. This means the counter associated with those writes must uniquely identify the read batch, not just the epoch. In fact, since every epoch has the same number of read batches, a read batch counter is sufficient for all values.

Handling Crashes The above modifications are sufficient to guarantee integrity if the proxy never crashes. When the proxy crashes, however, it needs information from the cloud storage to recover. To guarantee integrity—in particular freshness—of the recovery data, the epoch/read batch counter we describe above must persist in a trustworthy fashion across failures. Perhaps the easiest way to implement this requirement is to store the counter on a small amount of nonvolatile storage locally on the proxy, but any trustworthy and persistent storage mechanism is sufficient.

This, of course, raises the question of when to update this trustworthy persistent counter. Once the update occurs, a recovering proxy will expect the cloud storage to provide data associated with that counter value. This means that the counter must be updated after writing to cloud storage. Because a recovering proxy will be unaware of the newly-written data until the counter is updated, we do not consider the write complete until the counter is properly updated. As usual, if the proxy crashes while a write is in-progress, the write is simply rolled back.

As long as the storage server cannot learn anything from incomplete writes, our new strategy is entirely secure. Because the timing of Obladi's writes is completely deterministic and their locations are determined entirely by the locations of prior reads, the fact that a write has aborted does not inherently leak any information. The contents of the write can, however, leak information if we are not careful. Most data in the system is already encrypted, but one value is not: the read logs written during read batches. Previously we had no need to encrypt these as the write operation was atomic and the cloud server was to immediately learn all data contained in the write. However, the write is no longer atomic; the proxy can crash after sending data to the cloud server but before updating its trusted counter. In this case the storage server may withhold that data on recovery without detection and learn whether the proxy accessed the same locations after recovery is complete. To fix this leak, we encrypt the read batch logs in the cloud and update the counter after writing the log but before reading any values. That way the cloud storage gains no information about what data will be read until after the write is complete, at which point the proxy will

always replay the read if a crash occurs. Thus we have removed the leakage.

As we will see in Appendix B, these modifications are sufficient to guarantee both confidentiality and integrity (though obviously not availability) even against an arbitrarily malicious cloud storage server.

B Formal Security

We now provide formal security definitions and proofs for Obladi. As we discuss in §9, we use the Universal Composability (UC) framework [14]. The UC framework requires us to specify an ideal functionality $\mathcal{F}_{Ob}$ that defines what it means for Obladi to be secure. We must then prove that, for every possible adversarial algorithm $\mathcal{A}$ specifying the behavior of the storage server, we can simulate $\mathcal{A}$'s behavior when interacting only with $\mathcal{F}_{Ob}$.

We prove security of the scheme including the modification in Appendix A and do not assume the cloud storage provider is trusted for integrity. As the MACs and counters are only used to verify integrity and freshness of data, they are unnecessary if the cloud server is being honest. As we will see below, removing them—as we do in our implementation—does not impact security in this case.

We also noted in Appendix A that the proxy requires a trusted epoch counter that persists across crashes. This could be implemented as an integer in local non-volatile storage that the proxy updates with each epoch, it could be implemented by trusting the cloud storage for integrity and saving it there, or other means. We abstract away this detail by providing the Obladi protocol with access to $\mathcal{F}_{epc}$, an ideal functionality that provides access to this counter.

B.1 Ideal Functionality

We begin by noting that Obladi's proxy acts as a trusted central coordinator that performs publicly-known logic on private data. As this is essentially the role played by any ideal functionality, we simply subsume the proxy into $\mathcal{F}_{Ob}$. Moreover, some of the proxy's behavior, like the fact that it deduplicates and caches accesses, pads under-full batches, is public information, meaning $\mathcal{F}_{Ob}$ can explicitly perform exactly the same operations.

In §5 we describe the proxy as consisting of a concurrency control unit and a data manager, which itself contains a batch manager and ORAM executor. As the concurrency control and batch management functionalities do not inherently leak any information, we define $\mathcal{F}_{Ob}$ in terms of those operations. In particular, we let $\mathcal{F}^*_{Proxy}$ represent this functionality. $\mathcal{F}^*_{Proxy}$ is defined as providing the exact functionality of the concurrency control unit and batch manager as described in §5 and §6. $\mathcal{F}^*_{Proxy}$ has the following ways to interface with $\mathcal{F}_{Ob}$:

- $\mathcal{F}_{Ob}$ can supply $\mathcal{F}^*_{Proxy}$ with an input from a client (start, read, write, or commit).
- $\mathcal{F}^*_{Proxy}$ can produce a read batch of logical data blocks. The batch need not be full, meaning it may contain fewer than the maximum number of reads for a batch. $\mathcal{F}_{Ob}$ can then respond with the requested blocks.
- $\mathcal{F}^*_{Proxy}$ can produce a write batch of logical data blocks. The batch need not be full. $\mathcal{F}_{Ob}$ can then respond confirming the writes have completed.
- $\mathcal{F}^*_{Proxy}$ can specify an epoch has ended and transactions should commit. $\mathcal{F}_{Ob}$ can then respond with confirmation.
- $\mathcal{F}_{Ob}$ can clear $\mathcal{F}^*_{Proxy}$'s internal state, representing a crash.

$\mathcal{F}^*_{Proxy}$ can additionally send a messages directly to clients.

Modeling Crashes In the real system the proxy can crash at any time. As all state except the cryptographic keys (and possibly trusted counters) is considered volatile, it does not matter when during a local operation the proxy crashes, as every piece of that operation is lost regardless. We can therefore simplify the ideal functionality by allowing for crashes both between requests and immediately prior to any operation within a request that either leaves the proxy (e.g., writing to cloud storage) or persists across crashes (e.g., updating the trusted epoch counter).

To model any possible crash, we control the timing of the crashes through a Crash Client. $\mathcal{F}_{Ob}$ queries the Crash Client immediately prior to any relevant action and waits for a reply. The Crash Client then waits for a prompt from the environment, which it forwards to $\mathcal{F}_{Ob}$, telling it to proceed or crash. Additionally, the Crash Client—again at the prompting of the environment—can issue a "crash" command independently between requests.

We provide the full specification for $\mathcal{F}_{Ob}$ in Algorithm 1, which references $\mathcal{F}^*_{Proxy}$. For notational clarity, we do not explicitly specify every call to the Crash Client. Instead any operation prefixed by † notifies the Crash Client before executing and crashes if instructed. Note that it is possible to crash while recovering from a crash.

B.2 Security Lemmas

In order to prove the security of Obladi, we rely on two lemmas which we alluded to in §9.

Lemma 1 (Caching and Deduplication)**.** *Let D be any set of logical reads or writes selected independently from the current ORAM position map. Let D^* be the set of accesses resulting from applying the proxy batch manager's caching and deduplication logic to D. The set of physical accesses needed to realize D^* is identically distributed to the set of physical accesses needed to realize a uniformly random set of logical accesses of the same size.*

Proof. Since D is selected independently from the current position map in the ORAM, Ring ORAM guarantees that the set of physical accesses needed to realize D is

Algorithm 1: Ideal functionality $\mathcal{F}_{Ob}$ using $\mathcal{F}^*_{Proxy}$.

Data: D = DatabaseState
Data: Counters $c_e = 0$; $c_b = 0$

Initialize
  Initialize $\mathcal{F}^*_{Proxy}$
  Begin epoch
end

On receive m from client $\mathcal{C}$
  Forward $(m, \mathcal{C})$ to $\mathcal{F}^*_{Proxy}$
end

On receive "read-batch[*blks*]" from $\mathcal{F}^*_{Proxy}$
  †Send "read-batch-init" to $\mathcal{A}$, wait for "OK"
  †$c_b \leftarrow c_b + 1$
  †Send "read-batch-read" to $\mathcal{A}$, wait for "OK"
  Read *blks* from D
  Respond to $\mathcal{F}^*_{Proxy}$ with results
end

On receive "write-epoch[*data*]" from $\mathcal{F}^*_{Proxy}$
  †Send "write-epoch" to $\mathcal{A}$, wait for "OK"
  †$c_e \leftarrow c_e + 1$; $c_b \leftarrow 0$
  Write *data* to D
  Confirm write/epoch completed to $\mathcal{F}^*_{Proxy}$
end

On receive "crash" from Crash Client
  Execute `crashRecover`
end

function `crashRecover`
  Send ("crash", c_e, c_b) to $\mathcal{A}$
  Clear internal state of $\mathcal{F}^*_{Proxy}$
  Rollback writes to D since beginning of epoch c_e
  †$c_e \leftarrow c_e + 1$; $c_b \leftarrow 0$
end

†Before executing operation, notify Crash Client. On response of "crash," abort operation and invoke `crashRecover`, otherwise proceed.

identically distributed to that for a uniformly random set of logical reads or writes. D^* is simply D with some elements removed, so we claim that the elements removed form an unbiased sample. Since removing an unbiased sample from a distribution does not change the distribution, this is sufficient.

We first note that Ring ORAM guarantees that any independently-selected logical access d results in physical accesses sampled independently from the following distribution. First sample a uniformly random path in the tree. Then, for each bucket in that path, sample a uniformly random block from among those not read since the bucket was last written. Finally, read all selected blocks.

In Ring ORAM, whenever a block is read or written, it is immediately remapped to an independent uniformly random path in the tree that determines what will be read next time it is accessed. The proxy batch manger's caching and deduplication logic removes access requests for any block previously accessed in this epoch. Each of those blocks was mapped to a new independently uniform random path when accessed. Moreover, when an epoch ends, the cache is completely flushed, meaning there is no (potentially-biased) caching or deduplication.

Thus the sample of physical accesses removed by pairing D down to D^* must be unbiased, so D^* must result in a uniformly random set of physical access paths. □

Lemma 2 (Parallel ORAM)**.** *The set of parallel physical data operations performed by the proxy ORAM executor over one epoch (as described in §7) is completely determined by the set of sequential physical accesses required to perform the same logical actions in Ring ORAM (plus a single write to the durability store).*

Proof. We note that, as described in §7, the proxy performs all reads within an epoch before any writes (aside from the durability store). By construction, it ensures that each physical block that would be read at least once within an epoch in a fully sequential access is read exactly once in that epoch, and no other physical blocks are ever read (excluding crash recovery).

This is enforced by holding a record of every block that has been read this epoch and then performing the reads of the sequential access, but skipping blocks that have already been read. Additionally, whenever an evict path operation would happen, the proxy reads every unread block from each bucket along that path, thus marking them as read. As the timing of evict paths is determined by how many data accesses have happened and their locations are deterministic, this enforcement mechanism is dependent only on the physical blocks accessed, not in any way on the data held in those blocks.

Similarly, each block that would be written at least once in a sequentially-processed epoch is written exactly once at the end of the epoch. This is done by buffering writes in the proxy, allowing one buffered write of a physical block to overwrite any previous unflushed writes of that block. Then when the epoch ends, the proxy flushes all buffered writes. Again, the set of blocks being written is determined entirely by the physical access pattern of the sequential operation.

Finally, a fixed amount of data is written to the durability store before each read batch, and the entire durability store is written with each write batch. This means that in normal operation, the location and timing of all reads and writes are determined by only the physical operations needed to perform the epoch operations sequentially and some extra completely deterministic operations.

On crash recovery, the proxy reads the durability store and rereads all paths in the aborted epoch. This, again, is based entirely on physical access patterns.

Hence all physical read and write operations within a parallelized epoch are determined entirely by the physical data operations needed to perform that epoch sequentially. □

B.3 Proof of Security

We now prove that the Obladi protocol Π_{Ob} (with access to $\mathcal{F}_{epc}$) is secure with respect to the ideal functionality described in Algorithm 1. Let $\mathsf{Real}_{\mathcal{A},\mathcal{E}}(\lambda)$ denote the full transcript of $\mathcal{A}$ (including its inputs and randomness) when interacting with Π_{Ob}. Let $\mathsf{Ideal}_{\mathcal{S},\mathcal{E}}(\lambda)$ denote the transcript produced by $\mathcal{S}$ when run in the ideal world, interacting with $\mathcal{F}_{Ob}$.

Theorem 1. *Assume the encryption scheme used in Π_{Ob} is semantically secure and the MACs are existentially unforgeable. For all probabilistic polynomial time (PPT) adversaries $\mathcal{A}$ and environments $\mathcal{E}$, there is a simulator $\mathcal{S}_{\mathcal{A}}$ such that for all PPT distinguishers $\mathcal{D}$ there is some negligible function negl such that*

$$\left| \Pr\left[\mathcal{D}\Big(\mathsf{Real}_{\mathcal{A},\mathcal{E}}(\lambda)\Big) = 1\right] - \Pr\left[\mathcal{D}\Big(\mathsf{Ideal}_{\mathcal{S}_{\mathcal{A}},\mathcal{E}}(\lambda)\Big) = 1\right] \right| \leq negl(\lambda).$$

Proof. This proof follows from a series of hybrid simulators, each of which is indistinguishable from the previous.

We define hybrids $H_0, \ldots, H_4$. H_0 operates in the real world with $\mathcal{S}_0$ being a "dummy" that passes all messages through to $\mathcal{A}$ unmodified. H_1 has two ORAMs that are identical except for the MACs, one maintained by $\mathcal{A}$ and the other maintained by $\mathcal{S}_1$. H_2 replaces all data in $\mathcal{A}$'s ORAM with random dummy data, independent from the actual data. H_3 replaces the access pattern in $\mathcal{A}$'s ORAM with random data accesses. Finally H_4 uses $\mathcal{S}_{\mathcal{A}}$ in the ideal world and no longer maintains its own ORAM.

Hybrid H_0 contains a dummy simulator that passes messages between $\mathcal{A}$ and the proxy unchanged. This produces a transcript identical to the real world.

Hybrid H_1 passes all messages through to $\mathcal{A}$, but also maintains its own copy of the ORAM, simultaneously processes requests internally. On initialization $\mathcal{S}_1$ generates its own MAC key according to the same distribution as Π_{Ob}'s MAC key. It then replaces the MACs of all data sent to $\mathcal{A}$ with valid MACs on the same data using this new key. When $\mathcal{A}$ responds to a request, $\mathcal{S}_1$ checks the MACs on the data. If they are correct, it forwards the (correct) response from it's own ORAM with the original MACs. If they are incorrect, it responds with a failure message. If $\mathcal{A}$'s response is correct, so too will $\mathcal{S}_1$'s. If $\mathcal{A}$'s MACs to not verify, Π_{Ob} fails, so a failure message produces the same result. If $\mathcal{A}$'s response is wrong but the MACs verify, $\mathcal{A}$ must have forged a MAC since they include the data, position, and epoch counter, and no two pieces of data are ever given the same position and epoch counter. Moreover, because Π_{Ob} has access to a trusted epoch counter via $\mathcal{F}_{epc}$, it can properly verify that the data has the correct epoch counter, even after crashes. Thus, if $\mathcal{S}_1$ accepts an incorrect response with non-negligible probability, we can simulate $\mathcal{A}$ to forge a MAC with non-negligible probability. Hence H_1 is computationally indistinguishable from H_0.

Note that the MACs are only used to check that $\mathcal{A}$ provided correct data. If the storage server is assumed to be honest, this will always be the case and we can eliminate the MACs entirely (and also H_0 and H_1 become identical).

Hybrid H_2 replaces all data blocks provided to $\mathcal{A}$ with valid encryptions of random data and MACs on those encryptions. It otherwise passes on requests, including the location and timing of reads and writes. $\mathcal{S}_2$ continues to furnish responses to the proxy's queries using its internal ORAM with the original data, checking MACs according to the same scheme as in H_1. $\mathcal{S}_2$ then output's $\mathcal{A}$'s transcript. As all data is encrypted, the only difference between H_1 and H_2 is the contents of the ciphertexts, and by assumption the encryption scheme is semantically secure. This means H_1 and H_2 must be computationally indistinguishable.

Hybrid H_3 replaces all data requests to $\mathcal{A}$ with properly-formatted requests for randomly chosen data.

When $\mathcal{S}_3$ receives a location log for a read batch, it logs an encryption of random (unrelated) data with $\mathcal{A}$. When $\mathcal{S}_3$ receives the read instruction for a read batch, it first selects a random set of dummy paths of the batch size. It then requests $\mathcal{A}$ perform the proper parallel read operation for that dummy data. If $\mathcal{A}$ replies with the data and the MACs verify, $\mathcal{S}_3$ performs the actual reads on its separate ORAM with real data and returns the real data to the proxy.

When $\mathcal{S}_3$ is notified of the end of an epoch and given the associated write batch, it determines which physical blocks to write using Ring ORAM's deterministic write sequence based on the total number of operations (both reads and writes) in an epoch. It then performs proper parallel writes of new encryptions of dummy data to each of those locations. If $\mathcal{A}$ replied with confirmed writes, $\mathcal{S}_3$ performs the originally-specified operations on its separate ORAM and confirms success to the proxy.

Finally, if $\mathcal{S}_3$ receives a request to handle a proxy crash at epoch c_e and batch c_b, it queries $\mathcal{A}$ as per the crash recovery protocol for that epoch and batch. When $\mathcal{A}$ provides valid (MAC-verifying) read path logs for any batches this epoch, $\mathcal{S}_3$ provides the associated logs to the proxy. When the proxy issues redo read requests, $\mathcal{S}_3$ issues the same requests it did the first time to $\mathcal{A}$ for the associated batches.

Because $\mathcal{S}_3$ did not crash, it is able to retain which paths were read without having to store them explicitly. In is possible that the last read batch requested during recovery corresponds to a read that was never executed, in which case $\mathcal{S}_3$ generates a new random read batch and executes that instead. If $\mathcal{A}$ responds correctly, $\mathcal{S}_3$ responds to the proxy's requests.

By Lemma 1, the physical operations needed to process all real requests in a given epoch sequentially form an identical distribution to the sequential accesses needed to process the random requests chosen by $\mathcal{S}_3$. By Lemma 2, applying the parallelization process relies only on the sequential physical access pattern, meaning it can be applied the same way to $\mathcal{S}_3$'s random operations as to the real operations provided by the proxy. This means that the operations $\mathcal{S}_3$ requests of $\mathcal{A}$ are identically distributed to those the proxy requests of $\mathcal{S}_3$ when there are no crashes.

When a crash occurs, the recovery procedure is guaranteed to reread all previously-read data, and any future reads must have independently random paths. This is because $\mathcal{S}_3$ does not even generate random paths to read until the read request is issued, by which point the persistent batch counter c_b is updated. So if a crash does occur, it will redo any previous reads and future operations are treated as regular read/write batches with the same (independent) distribution. Since these are the only difference between H_2 and H_3, the two must produce identical distributions.

Hybrid H_4 now interacts with the ideal functionality and no longer maintains its own internal ORAM copy, only the data necessary to perform actions on $\mathcal{A}$'s, including the new MAC and encryption keys. The only data $\mathcal{S}_3$ was using to compute requests for $\mathcal{A}$ was the timing of batches and crash recoveries, and the epoch and batch counters during recovery. As $\mathcal{F}_{Ob}$ explicitly provides all of that information, $\mathcal{S}_4$ is able to provide $\mathcal{A}$ with an identical view. Note that on crash recovery, this identical view requires completing a crash-recover epoch, which $\mathcal{S}_4$ can do by creating an appropriate number of read and write operations as it would in H_3. This means that H_3 and H_4 are identically distributed.

Thus we see that H_0 corresponds to the real world, H_4 corresponds to the ideal world, and each sequential pair of (H_i, H_{i+1}) produce computationally indistinguishable transcripts. Thus it must be the case that H_0 and H_4 form computationally indistinguishable transcripts, so Π_{Ob} realizes $\mathcal{F}_{Ob}$. $\square$