ObliviSync: Practical Oblivious File Backup and Synchronization

Adam J. Aviv* Seung Geol Choi* Travis Mayberry† Daniel S. Roche*

*Computer Science Department
†Cyber Science Department
United States Naval Academy
Annapolis, Maryland, U.S.A.

May 30, 2016

Abstract

Oblivious RAM (ORAM) is a powerful cryptographic protocol which has recently risen in prominence, in part due to its ability to hide a client's access patterns from untrusted cloud storage services. We present an oblivious cloud storage system, ObliviSync, that specifically targets one of the most widely-used personal cloud storage paradigms: synchronization and backup services, popular examples of which are Dropbox, iCloud Drive, and Google Drive. We show that this setting provides a unique opportunity for Oblivious RAM research because full privacy can be achieved with a simpler form of ORAM called *write-only* ORAM. This allows for dramatically increased efficiency compared to related work — so much so that our solution has only a small constant overhead of approximately 4x compared with non-private file storage. We built and evaluated a full implementation of ObliviSync that supports multiple simultaneous read-only clients and a single concurrent read/write client whose edits automatically and seamlessly propagate to the readers. We show that our system functions under high work loads with realistic file size distributions.

1 Introduction

ORAM: security and efficiency. ORAM is a protocol which allows a client to access files (commonly abstracted as N fixed-length *blocks* of data) stored on an untrusted server in such a way that the server learns neither the *contents* of files nor the *access patterns* of which files were accessed at which time(s). This is traditionally accomplished by doing some type of shuffling on the data in addition to reading/writing the chosen block. This shuffling ensures that the server cannot correlate logical blocks based on their storage locations.

ORAM is a powerful tool that solves a critical problem in cloud security. Consider a hospital which uses cloud storage to backup their patient records. Even if the records are properly encrypted, an untrusted server that observes which patient files are modified will learn sensitive medical information about those patients. They will certainly learn that the patient has visited the hospital recently, but also may learn things like whether the patient had imaging tests done based on how large the file is that is updated. Moreover, they might learn for instance that a patient has cancer after seeing an oncologist update their records. This type of inference, and more, can be done despite the fact that the records themselves are encrypted because the *access pattern* to the storage is not hidden.

Unfortunately, in order to achieve this obliviousness ORAMs often require a substantial amount of shuffling during every access, so much so that even relatively recent ORAM constructions could

induce a several-thousand-fold overhead on communication [18, 15]. Even Path ORAM [19], one of the most efficient ORAM constructions to date, has a practical overhead of 60-80x on moderately sized databases compared to non-private storage.

Our goal. In this paper, we present an efficient solution for oblivious storage on a personal cloud synchronization/backup provider such as (but not limited to) Dropbox or Google Drive.

The setting: personal cloud storage. Our setting consists of an untrusted cloud provider and one or more clients which backup data to the cloud provider. If there are multiple clients, the cloud provider propagates changes made by one client to all other clients, so that they each have the same version of the filesystem. We emphasize that although we may use "Dropbox" as a shorthand for the scenario we are addressing, our solution is not specific to Dropbox and will work with any similar system. This setting is particularly interesting for a number of reasons:

1. It is one of the most popular consumer cloud services used today, and is often colloquially synonymous with the term "cloud". Dropbox alone has over 500 million users [11].
2. The interface for Dropbox and similar storage providers is "agnostic", in that it will allow you to store any data as long as you put it in the designated synchronization directory. This allows for one solution that works seamlessly with all providers.
3. Synchronization and backup services do not require that the ORAM hide a user's read accesses, only the writes.

Write-only ORAM. The last point in the above (i.e., we don't need to hide read accesses) is crucial to the efficiency of our system. Each client already has a copy of the database, so when they read from it they do not need to interact with the cloud provider at all. If a client writes to the database, the changes are automatically propagated to the other clients with no requests necessary on their part. Therefore, the ORAM protocol only needs to hide the write accesses done by the clients and not the reads. This is important because [8] have shown that *write-only* ORAM can be achieved with optimal asymptotic communication overhead of $O(1)$. In practice, write-only ORAM requires only a small constant overhead of 3-6x compared to much higher overheads for fully-functional ORAM schemes, which asymptotically are $\Omega(\log N)$.

Supporting variable-size files. When addressing a personal cloud setting, a crucial aspect that must be dealt with is the variable sizes of the files stored in such a system. Traditionally, ORAMs are modeled as storage devices on N fixed-length blocks of data, with the security guarantee being that any two access patterns of the *same length* are indistinguishable from each other. In reality, files stored on Dropbox are of varying (sometimes unique) lengths. This means that a boilerplate ORAM protocol will actually not provide obliviousness in such a setting because the file size, in multiples of the block size, will be leaked to the server for every access. When file sizes are relatively unique, knowing the size will enable the server to deduce which individual file is being accessed, or at least substantially reduce the number of possibilities. Therefore our solution additionally includes a mechanism for dynamically batching together variable-length files to hide their size from the server. Furthermore, our solution is *efficient* as we prove its cost scales linearly with the total size (and not number) of files being written, regardless of the file size distribution. Conveniently, the batching aspect of our construction also allows us to protect against *timing-channel attacks*, which are not usually considered in ORAM protocols.

Providing cloud-layer transparency. Additionally, one of the most noteworthy aspects of Dropbox-like services is their ease of use. Any user can download and install the requisite software,

at which point they have a folder on their system that "magically" synchronizes between all their machines, with no additional setup or interaction from the user. In order to preserve this feature as much as possible, we implement our system as a FUSE filesystem that mounts on top of the shared directory, providing a new directory where the user can put their files to have them *privately* stored on the cloud. The FUSE module uses the original shared directory as a backend by storing fixed-size blocks as individual files. Since ORAMs only traditionally support storing N blocks indexed by the numbers $[0, N)$, substantial work is needed to add support for filesystem features like filenames and sizes. Our implementation, then, necessarily consists of an ORAM algorithm merged with traditional filesystem concepts, including i-/v-nodes, superblocks, etc.

Summary of our contribution. To summarize, our contributions in this paper include:

1. A complete ORAM system designed for maximum efficiency and usability when deployed on a synchronization/backup service like Dropbox.
2. A proof of strong security from an untrusted cloud provider, even considering the timing side-channel.
3. A FUSE implementation of these contributions, incorporating variable size files as well as important filesystem functionality into ORAM including the ability to store file names, resize files and more.
4. Theoretical evaluation showing that our scheme requires only 4x bandwidth overhead compared to unencrypted and non-private storage, regardless of the underlying file size distribution. We also show that our scheme has very high storage utilization, requiring only 1.5-2.0x storage cost overhead in practice.

2 Efficient Oblivious Dropbox

2.1 Overview of Write-only ORAM

We start by describing the write-only ORAM of [8], as it informs our construction.

The setting. To store N blocks in a write-only ORAM, the server holds an array of $2N$ encrypted blocks. Initially, the N blocks of data are shuffled and stored in random locations in the $2N$-length array, such that half of the blocks in the array are "empty". However, every block is encrypted with an IND-CPA encryption scheme so the server cannot learn which blocks are empty and which are not. The client stores a *local dictionary* (or sometimes called a position map) which maps a logical address in the range $(0, N]$ to the location in the server array where it is currently stored, in the range $(0, 2N]$. Using this dictionary, the client can find and read any block in the storage that it needs, but the server will not know the location of any individual block.

Read and write operations. Since by definition a write-only ORAM does not need to hide reads, they are performed trivially by reading the local dictionary and then the corresponding block from the ORAM. Write operations, however, require additional work. When the client wishes to write a block to the ORAM, it chooses k random locations in the array out of $2N$, where k is a constant parameter. Out of these k locations, it chooses one which is empty and writes the new block into that location, while reencrypting the other $k-1$ locations to hide from the server which block was changed. After writing the block, the client also updates their dictionary to indicate that the block now resides in its new location. The old location for this block is implicitly marked empty because no entry in the dictionary now points to it.

Achieving obliviousness. Since every write operation sees the client accessing k randomly chosen blocks in the ORAM, independent of the logical address of the block that is being written, it cannot reveal any information about the client's access pattern. The only situation that can cause the client to reveal something is if the k chosen locations do not contain any free blocks, and it has nowhere to write the new one. Since every block has $1/2$ probability of being empty, the chance that there are no free blocks will be 2^{-k}, so k can be set to the security parameter λ to give a negligible chance of failure.

Efficiency with stash on the client. However, setting $k = \lambda$ actually does not result in $O(1)$ overhead; since $\lambda > \log N$, the overhead is $\Omega(\log N)$. On average, the client find $k/2$ empty blocks during a single write, many more than are needed. If the client instead stores a buffer of blocks that it wants to write, and writes as many as he finds empty blocks for, k can be set much more aggressively. It is shown in [8] that $k = 3$ is sufficient to guarantee with high probability that the stash will never exceed $O(\log N)$. This makes the final overhead for an ORAM write only 6x that of non-private storage (3x for reading the k blocks and 3x for writing them back).

Maintaining the dictionary file. The final important detail is that the dictionary file requires $O(N \log N)$ bits of storage, which might be too large for the client to store locally. Fortunately it is relatively simple to store this dictionary recursively in another ORAM [8, 19]. For some block and databases sizes, however, it might be quite reasonable for the client to store the entire dictionary itself.

2.2 Overview of Our System

The setting. Our ObliviSync system uses the idea of write-only ORAM on top of any file backup or synchronization tool in order to give multiple clients simultaneous updated access to the same virtual filesystem, without revealing anything at all to the cloud service that is performing the synchronization itself. Write-only ORAM is ideal for this setting because *each client stores an entire copy of the data*, so that only the changes (write operations) are revealed to the synchronization service and thus only the write operations need to be performed obliviously.

Improvements over write-only ORAM. Compared to previous write-only ORAM construction [8], we make significant advances and improvements to fit this emergent application space:

- **Usability**: All users interact with the system as though it is a normal system folder. All the encryption and synchronization happens automatically and unobtrusively.
- **Flexibility**: We support a real filesystem and use innovative methods to handle variable-sized files and changing client roles (read/write vs. read-only) to support multiple users.
- **Strong obliviousness**: The design of our system not only provides obliviousness in the traditional sense, but also *protects against timing channel attacks*. It also conceals the total number of write operations, a stronger guarantee than previous oblivious proposals.
- **Performance**: Our system well matches the needs of real file systems and matches the services provided by current cloud synchronization providers. It can also be tuned to different settings based on the desired communication rate and delay in synchronization.

Basic architecture. The basic design of ObliviSync is presented in Figure 1. There are two types of clients in our system: a read/write client (ObliviSync-RW) and a read-only client (ObliviSync-RO). There can be many ObliviSync-RO's active at one time on the same filesystem, but only at most

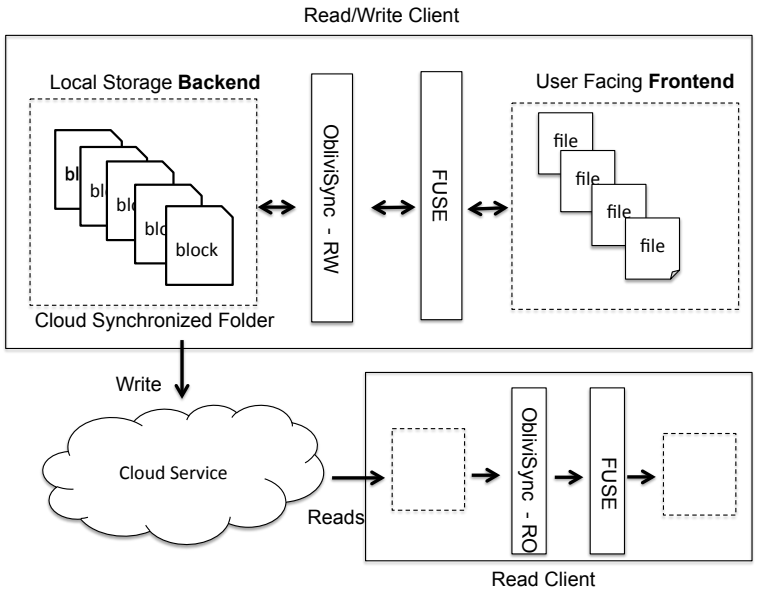

Figure 1: Diagram for ObliviSync

one ObliviSync-RW. Both clients consist of an actual *backend* folder as well as a virtual *frontend* folder, with a FUSE client running in the background to seamlessly translate the encrypted data in the backend to the user's view in the frontend virtual filesystem.

We rely on existing cloud synchronization tools to keep all clients' backend directories fully synchronized. This directory consists of encrypted files that are treated as generic storage blocks, and embedded within these storage blocks is a file system structure loosely based on the i-node style file systems which allows for variable-sized files to be split and packed into fixed-size units. Using a shared private key, the job of both clients ObliviSync-RO and ObliviSync-RW is to decrypt and efficiently fetch data from these encrypted files in order to serve ordinary read operations from the client operating in the frontend directory.

The ObliviSync-RW client, which will be the only client able to change the backend files, has additional responsibilities: (1) to maintain the file system encoding embedded within the blocks, and (2) to perform updates to the blocks in an oblivious manner using our efficient modification of the write-only ORAM described in the previous subsection.

User transparency with FUSE mount. From the user's perspective, however, the interaction with the frontend directory occurs as if interacting with any files on the host system. This is possible because we also implemented a FUSE mount (file system in user space) interface which displays the embedded file system within the blocks to the user as if it were any other file system mount. Under the covers, though, the ObliviSync-RO or ObliviSync-RW clients are using the backend directory files in order to serve all data requests by the client, and the ObliviSync-RW client is additionally monitoring for file changes/creations in the FUSE mount and propagating those changes to the backend.

Strong obliviousness through buffered writes. In order to maintain obliviousness, these updates are *not* immediately written to the backend filesystem by the ObliviSync-RW client. Instead, the process maintains a *buffer* of writes that are staged to be committed. At regular timed intervals, random blocks from the backend are loaded, repacked with as much data from the buffer as possible, and then re-encrypted and written back to the backend folder. From there, the user's chosen file

synchronization or backup service will do its work to propagate the changes to any read-only clients. Crucially, as the number of blocks written at each step is fixed, and these writes occur at regular timed intervals, an adversary operating at the network layer is unable to determine anything about the file contents or access patterns.

Further details on all of these components can be found in Section 4. The full source code of our implementation is also available upon request.

3 Security Definitions

3.1 Write-only Oblivious Synchronization

Block-based filesystem. Our system has more capabilities than a standard ORAM, including support for additional filesystem operations, so we require a modified security definition which we present here. We first formally define the syntax of a block-based filesystem with block size B.

- create(filename): create a new (empty) file.
- delete(filename): remove a file from the system.
- resize(filename,size): change the size of a file. The size is given in bytes, and can be greater or smaller than the current size.
- write(filename,offset,length): write data to the identified file according to the offset and length arguments. The offset is a *block offset*. Unless the offset refers to the last block in the file, length must be a multiple of B.
- read(filename,offset,length) $\rightarrow$ data: read data from the identified file according to the offset and length arguments. Again, offset is a block offset, and length must be a multiple of the block size B unless the read includes the last offset.

For simplicity, we only consider these five core operations. Other standard filesystem operations can be implemented using these core functionalities.

Obliviousness and more. The original write-only ORAM definition in [8] requires indistinguishability between any two write accesses with *same data sizes*. However, the definition does not consider the time at which write operations take place. Here, we put forward a stronger security notion for the file system that additionally hides both the data length and the time of non-read operations.

For example, we want to make sure all the following operation sequences are indistinguishable:

- no write at all
- write(file1,1,5) and write(file2,3,3) at time 3, and
  write(file1,6,9) at time 5
- write(file2,1,20) bytes at time 5

For this purpose, we first define (L, t)-fsequences. Here, the parameter L is the maximum number of bytes that may be modified (whether filesystem metadata or actual file data), and t is the latest time that is allowed. For example, the above sequences are all $(20, 5)$-fsequences, since all sequences write at most 20 bytes and have the last write before or at time 5.

Definition 1 (**(L,t)-fsequence**). *A sequence of non-read operations for a block filesystem is a* (L,t)-fsequence *if the total number of bytes to be modified in the filesystem metadata and file data is at most L, and the last operation takes place before or at time t.*

Our goal is to achieve an efficient block filesystem construction such that any two (L,t)-fsequences are indistinguishable.

Definition 2 (**Write-only strong obliviousness**). *Let L and t be the parameters for fsequences. A block filesystem is* write-only strongly-oblivious with running time *T, if for any two (L,t)-fsequences P_0 and P_1, it holds that:*

- *The filesystem finishes all the tasks in each fsequence within time T with probability $1-neg(\lambda)$, where is λ is the security parameter.*
- *The access pattern of P_0 is computationally indistinguishable to that of P_1.*

4 System Details

As described previously, the basic design of ObliviSync is presented in Figure 1. In this section we highlight the implementation details further. In the rest of this section, we describe the implementation components in more details focusing on interesting design challenges and user settings that can be used to the tune the performance.

4.1 Filesystem Description

We describe how we organize the data in the backend files to provide a general filesystem. We note that our filesystem is *specifically tailored for the ObliviSync use case*, and this design is what leads to our practical performance gains.

Terminology. The terms file, fragment, and block are used throughout this section. The user of an ObliviSync client is creating, writing, reading, and deleting *files* in the frontend filesystem via the FUSE mount. The ObliviSync client, to store user files, will break down the files into one or more *fragments*, and these fragments are then stored within various encrypted *blocks*, stored in pairs as a backend file. Recall that it is only these encrypted blocks which are seen (and transferred) by the cloud synchronization service. While each block has the same size, files stored in the frontend can have arbitrary sizes. A file fragment can be smaller than a block size, but not larger.

V-nodes and i-nodes. V-nodes and i-nodes are used (respectively) to identify a single file in the frontend, and a single block in the backend[1]. In particular, the v-nodes can be assigned arbitrarily as they only exist "virtually" to identify files in the user frontend, whereas the i-nodes for blocks correspond to actual named files of encrypted blocks in the backend directory.

A single file consists of a series of fragments (or a single fragment if the file is small), where each fragment is stored within a block. Since frontend, user-facing files are identified using v-nodes and backend blocks with i-nodes, a *v-node table* is necessitated to map every v-node to a list of i-nodes to refer to the blocks that contain the file's fragments, in order. As files update, the v-node remains constant; however, based on the oblivious writing procedure, the i-nodes may change. Further, multiple v-nodes may share the same i-node if the file fragments are less than a block size.

[1]The terms `vnode` and `inode` have a related, but not identical meaning, in Linux filesystem implementations. The reader should not assume any particular meaning from the use of these terms, other than these are numbers which identify files in some way.

To facilitate this, some structured metadata must also be stored within each block, as well as the v-node table itself. Additionally, each v-node table entry also stores the size of the corresponding frontend file and the last access time.

Superblock. The v-node table that maps v-nodes to i-node lists is implemented as a B-tree. The root of the v-node table B-tree is *always* stored within the block at index 0, which we refer to as the *superblock*.

In general, the height of this B-tree is $O(\log_B n)$, where B is the number of bytes in a single block and n is the number of files stored. As we will see in Section 4.6, for typical scenarios, the block size B is sufficiently large such that the B-tree height will always be at most 1, and we will assume that $\log_B n \leq 1$ for the remainder.

In fact, if B is sufficiently large relative to n, the B-tree will have height 0 and we can store the entire v-node table in the superblock. When the height of the B-tree equals 1, the leaf nodes are added and stored in ordinary blocks, indexed by the i-node locations. Note that this differs from ordinary files in that they are indexed by i-nodes which change every time a child node is re-written. This is allowable because the root node is part of the superblock which is re-written at every step (as we will see) and because the B-tree is configured specifically so that each leaf will fit within a single block.

To further improve performance and avoid frequent writes to the B-tree leaves, we also maintain a *v-node cache* within the superblock which tracks the most recent mapping of v-node to its list of i-nodes. This is particularly relevant when there are large number of files in the filesystem such that we can avoid extra accesses (or stage updates to) B-tree leaves for successive accesses to the same file.

In total, the superblock contains the parameters of the ObliviSync (e.g., the block size), the v-node cache, and the root node of the v-node table. Additionally, we ensure that the root directory is always stored at v-node 0, and directories are stored as ordinary files containing the names and v-nodes of their contents as is typical in most filesystems.

Split vs. full blocks. At the block level, there can be two types of blocks: a *full block* where the fragment stored within is as large as the block size and inhabits the entirety of the block, or a *split block* where multiple fragments smaller than the block size are stored within the same block. When a large file is stored as a series of fragments, we maintain that all fragments *except possibly for the last fragment* are stored in full blocks. That is, there will be at most one split-block fragment per file.

Looking up the data in a full block is straightforward: given the i-node value, ObliviSync fetches the given block and decrypts its contents. In addition to the actual data, we also store the v-node of the file within the block itself as metadata. This will facilitate an easy check for whether a given block has become *stale*, as we will see shortly.

For a split block, however, the system also needs to know the location of the desired fragment within the block. The information is stored within the block itself in the *block table* which maps v-nodes to offsets. With the size of the file from the superblock, it is straightforward to retrieve the relevant data. A full block can then be simply defined as a block *without* a block table, and the leading bit of each block is used to identify whether the block is full or split.

Two blocks stored in each backend file. One additional detail, very important to the performance of ObliviSync, is that all blocks on the backend are grouped into *pairs* of two consecutive blocks where each pair of blocks resides within a single file in the backend directory. Crucially, *small fragments are allowed to reside in either block in the pair* without changing their i-node. Furthermore, as we will see in the sync operation described later, both blocks in a given pair are

randomly selected to be re-packed and rewritten at the same time. This additional degree of freedom for small file fragment is crucial for reducing the worst-case efficiency of the system where there only exist many very small files.

4.2 Read-Only Client.

A read-only client (ObliviSync-RO) with access to the shared private key is able to view the contents of any directory or file in the frontend filesystem by reading (and decrypting) blocks from the backend, but cannot create, destroy, or modify any file's contents.

To perform a read operation for any file, given the v-node of that file (obtained via the directory entry), the ObliviSync-RO first needs to obtain the i-node list of the file's fragments to then decrypt the associated blocks and read the content. This is accomplished in the following steps:

1. Decrypt and read the superblock.
2. Check in the v-node cache. If found, return the corresponding i-node list.
3. If not found in cache, search in the v-node table via the B-tree root (part of the superblock) to find the i-node location of the appropriate leaf node in the B-tree.
4. Decrypt and read the leaf node to find the v-node table entry for the file in question, and return the corresponding i-node list and associated meta-data.

Once the i-node list has been loaded, the desired bytes of the file is loaded by computing the i-node offset according to the block size, loading and decrypting the block specified by that i-node, and extracting the data in the block matching the file's v-node.

Given the v-node, it can be seen from the description above that a single read operation in ObliviSync-RO for a single fragment requires loading and decrypting at most 3 blocks from the backend: (1) the superblock, (2) a B-tree leaf node, and (3) the block containing the data. In practice, we can cache recently accessed blocks (most notably, the superblock) for a short period in order to speed up subsequent lookups.

4.3 Read/Write Client

The read/write client ObliviSync-RW encompasses the same functionality as ObliviSync-RO for lookups with the added ability to create, modify, and delete files.

The additional data structure stored in ObliviSync-RW to facilitate these write operations is a *buffer* of recent, un-committed changes. Specifically, this buffer stores a list of (v-node, fragment, timestamp) tuples. When the ObliviSync-RW encounters a write (or create or delete) operation, the operation is performed by adding to the buffer. For modified files that are larger than a block size, only the fragments of the file that need updating are placed in the buffer, while for smaller files, the entire file may reside in the buffer.

During reads, the system first checks the buffer to see if the v-node is present such that the freshest data is retrieved. Otherwise, the normal read operation as in the ObliviSync-RO description above is used.

The main motivation of the buffer is to allow obliviousness writing without compromising usability. The user should not be aware of the delay between when a write to a file occurs and when the corresponding data is actually synced to the cloud service provider. The function of the buffer is similar to that of stash in normal ORAM constructions.

Interestingly, we note that the buffer also provides considerable *performance* benefits, by acting as a cache for recently-used elements. Since the buffer contents are stored in memory un-encrypted, reading file fragments from the buffer is faster than decrypting and reading data from the backend storage. The buffer serves a dual purpose in both enabling obliviousness and increasing practical efficiency.

4.4 Syncing the buffer to the backend.

The buffer within ObliviSync-RW must not grow indefinitely. In our system, the buffer size is kept low through the use of a periodic *sync* operations wherein the buffer's contents are encrypted and stored in backend blocks.

Each sync operation is similar to a single write procedure in write-only ORAM, but instead of being triggered on each write operation, the sync operation happens on a *fixed timer* basis. We call the time between subsequent sync operations an *epoch* and define this parameter as the *drip time* of the ObliviSync-RW.

Also, similar to the write-only ORAM, there will be a fixed set of blocks that are rewritten on each sync operation epoch. The number of such blocks that are rewritten and re-encrypted is defined as the *drip rate*. We discuss these parameters further and there impact on performance in Section 4.6.

Choosing which blocks to rewrite. As in write-only ORAM, k random locations are chosen to be rewritten at every sync operation, with the following crucial differences:

- Position 0, which stores the superblock, is *always* chosen to be rewritten.
- Pairs of blocks (as described above) are always chosen together, so that the $k-1$ random locations are actually $k-1$ random *pairs* of blocks to rewrite.

The superblock must be rewritten on each sync because it contains the v-node table which may change whenever other content is rewritten to the backend. Choosing pairs of blocks together is also crucial, since as we have mentioned above, small fragments are free to move between either block in a pair without changing their i-nodes.

Determining staleness. Once the blocks to be rewritten are randomly selected and decrypted, the next task is to inspect the fragments within the blocks to determine which are "stale" and can be overwritten.

Tracking fragment freshness is vital to the system because of the design of write-only ORAM. As random blocks are written at each stage, modified fragments are written to *new* block locations and update the v-node table entry, but the stale data fragment is *not* rewritten and will persist in the old block location because that old block may not have been selected in this current sync procedure. Efficiently identifying which fragments are stale becomes crucial to clearing the buffer.

A natural, but flawed, solution to tracking stale fragments is to maintain a bit in each block to mark which fragments are fresh or stale. This solution cannot be achieved for the same reason that stale data cannot be immediately deleted — updating blocks that are not selected in the sync procedure are not possible — but recall from the block design that each block also stores the v-node for each fragment. To identify a stale fragment, the sync procedure looks up each fragment's v-node to get its i-node list. If the current blocks identifier is not in the i-node list, then that fragment must be stale.

Re-packing the blocks. Then next step after identifying blocks and fragments within those blocks that are stale (or empty) is to *re-pack* the block with fragments from the buffer with fresh fragments residing within the block.

A naïve approach would be to evict all the non-stale fragments from the selected block and consider all the fragments and evicted fragments to re-pack the selected blocks with the least amount of internal fragmentation. While this would be a reasonable protocol for some file systems to reduce fragmentation; however, this would require (potentially) changing all of the v-node table entries for all fragments within the selected blocks. That would be problematic because it is precisely these old entries which are likely not to be in the v-node cache, and therefore doing this protocol would require potentially changing many v-node B-tree nodes at each step, something that should be avoided as writes are so expensive.

Instead, *existing full-block data does not change location, and existing split-block fragments may only move to the other block in the pair.* Recall that blocks are paired in order and share a single i-node, and so this flexibility enables small fragments to be re-packed across two blocks to reduce fragmentation without having to update the i-node value. Further, this solution also avoids a "full-block starvation" issue in which all blocks contained just a small split-block fragment. After re-packing the small fragments in each block pair will be combined into a single split block, leaving the other block in the pair empty and ready to store full block fragments from the buffer. In other words, the re-pack procedure ensures that existing full-block fragments do not move, but existing small-block fragments are packed efficiently within one of the blocks in a pair to leave (potentially) more fully empty blocks available to be rewritten.

At this point, the randomly-chosen blocks are each either: (a) empty, (b) filled with an existing full-block fragment, or (c) partially filled with some small fragments. The block re-packing then considers all fragments in the buffer in FIFO order, and for each fragment, it tries to pack it into the $k - 1$ randomly-selected blocks as follows:

- If the fragment is a full block, it is placed in the first available empty block (case (a)), if any remain.
- If the fragment is a split block, it is placed if possible in the first available split block (case (c)) where there is sufficient room.
- If a split block fragment cannot fit in any existing split block, the first available empty block (case (a)), if any remain, is initialized as a new split block containing just that fragment.

In this way, every buffer fragment is considered for repacking in order of age, but not all may actually be re-packed. Those that are will be removed from the buffer and their v-node table entries will be updated according to the chosen block's location.

A *key observation* that will be important in our runtime proof later is that after re-packing, either (1) there are no empty blocks out of the set that is being re-packed, or (2) the buffer is completely cleared.

After the re-packing is complete, the sync procedure re-encrypts the superblock (which always goes at index 0), as well as all the repacked blocks, and stages them for writing back to backend files. The actual writing is done all at once, on the timer, so as not to reveal how long the sync procedure took to complete.

Consistency and ordering. In order to avoid inconsistency, busy wait, or race conditions, the order of operations for the sync procedure is very important. For each file fragment that is successfully cleared from the buffer into the randomly-chosen blocks, there are three changes that must occur:

- The data for the block is physically written to the backend.

- The fragment is removed from the buffer.
- The v-node table is updated with the new location for that fragment.

It is very important that these three changes occur *in that order*, so that there is no temporary inconsistency in the filesystem. Moreover, the ObliviSync-RW must *wait until all fragments of a file have been synced* before updating the v-node entry for that file; otherwise there could be inconsistencies in any ObliviSync-RO clients.

One consequence of this ordering is that it will always take at least two sync stages for a single file's changes to be propagated to any ObliviSync-RO clients because the v-node table will never be updated until one stage *after* all the fragments themselves have been synced back. This adds a small degree of latency to the system, but the benefit is that both types of clients both have a clean, consistent (though possibly temporarily outdated) view of the filesystem.

4.5 Frontend FUSE Mounts

The FUSE (file system in user space) mounts are the primary entry point for all user applications. FUSE enables the capture of system calls associated with I/O, and for those calls to be handled by an identified process. The result is that a generic file system mount is presented to the user, but all access to that file system are handled by either the ObliviSync-RW or ObliviSync-RO client that is running in the background.

The key operations that are captured by the FUSE mount and translated into ObliviSync-RW or ObliviSync-RO calls are as follows:

- create($filename$) : create a new (empty) file in the system in two steps. First a new v-node is chosen and added to the v-node table. Then that v-node is also stored within the parent directory file.
- delete($filename$) : remove a file from the system by removing it from the directory file and removing the associated v-node from the v-node table.
- read($filename, offset, length$) $\rightarrow$ data : read data from the identified file by looking up its v-node in the directory entry and requesting the backend ObliviSync-RW or ObliviSync-RO to perform a read operation over the appropriate blocks.
- write($filename, offset, length$) : write data to the identified file by looking up its v-node in the directory entry and then adding the corresponding fragment(s) to the ObliviSync-RW's buffer for eventual syncing.
- resize($filename, size$) : change the size of a file by looking up its v-node in the directory entry and changing the associated metadata. This may also add or remove entries from the corresponding i-node list if the given size represents a change in the number of blocks for that file. Any added blocks will have `NULL` i-node values to indicate that the data is not yet available.

Of course, there are more system calls for files than these, such as `open()` or `stat()` that are implemented within the FUSE mount, but the above described functions compactly encodes all major operations between the frontend FUSE mount and backend file system maintenance.

As noted above, another task of the FUSE mount is to maintain the file system's directory structure whose main purpose is to link file names to their v-node values, as well as store other

expected file statistics. The directory entry, itself, though, is treated just like any file stored in the system, except that the root directory file is always assigned v-node 0.

While we designed our systems to provide separation between the frontend and backend, the FUSE process needs to be aware of some backend settings to improve performance, notably the block size. When a file is modified, it is tracked by the FUSE process, but for large files, with knowledge of the block size, the FUSE process can identify which full fragments of that file is modified and which remain unchanged. It will then only propagate which fragments of a large file changed to the backend such that synchronization only re-packs the contents of the file that actually changed.

4.6 Key parameter settings

Here, we bring together all the key settings for the ObliviSync and what typical ranges these parameters might take in a real implementation.

The first choices a user must in setting up ObliviSync is dependent on the backend cloud service: B, the size of each file in backend storage, and N, the total number of such files. A typical example of such parameters can be taken from the popular Dropbox service, which optimally handles data in files of size 4MB, so that $B = 2^{22}$ [10], and the maximal total storage for a paid "Dropbox Pro" account is 1TB, meaning $N = 2^{18}$ 4MB files would be the limit[2]

The next parameter, n, is the total number of files that the user will store within the ObliviSync file system. In fact this is not a parameter *per se* but rather a limitation, as our construction requires $n \leq B^2$ in order to ensure the v-node table's B-tree has height at most 1. For $B = 2^{22}$, this means the user is "limited" to roughly 16 trillion files.

There are two key settings for the buffer syncing: *drip rate* k and *drip time* t. The drip rate refers to how many block pairs are selected for rewriting on each epoch, and the *drip time* is the length of the epoch, i.e., the time between consecutive writes of k files to the backend.

These two parameters provide a clear trade-off between latency and throughput. Given a fixed bandwidth limitation of, say, x bytes per second, kB bytes will be written every t seconds, so that we must have $kB/t \leq x$. Increasing the drip time and drip rate will increase latency (the delay between a write in the ObliviSync-RW appearing to ObliviSync-RO clients), but increase throughput as the constant overhead of syncing the superblock happens less frequently. We will consider in our experimentation section (see Section 6) the throughput, latency, and buffer size of the system under various drip rate and drip time choices. Our experiment indicates that for most uses, the smallest possible drip time t that allows a drip rate of $k \geq 3$ files per epoch should be chosen.

5 Analysis

5.1 Time to write all files

We first remind the reader of the following notation for the ObliviSync parameters, which will be used throughout this section:

- B is the size (in bytes) of backend files, each of which is large enough to store 2 full "blocks" of data.
- N is the total number of size-B files in backend storage.

[2]Note that, as the our construction always writes blocks in pairs, each block block pair is stored in a single file and the block size in ObliviSync will be $B/2$.

- k is the "drip rate", i.e., the number of size-B files (excluding the superblock) that are written in each sync operation.

In this subsection we will prove the main Theorem 1 that shows the relationship between the number of sync operations, the drip rate, and the size of the buffer. Specifically, we will show that, with high probability, a buffer with size s is completely cleared and synced to the backend after $O\left(\frac{s}{Bk}\right)$ sync operations. This is optimal up to constant factors, since only Bk bytes are actually written during each sync.

Theorem 1. *For a running ObliviSync-RW client with parameters B, N, k as above, let m be the total size (in bytes) of all non-stale data currently stored in the backend, and let s be the total size (in bytes) of pending write operations in the buffer, and suppose that $m + s \leq NB/5$.*

Then the expected number of sync operations until the buffer is entirely cleared is at most $4s/(Bk)$.

Moreover, the probability that the buffer is not *entirely cleared after at least*

$$\frac{16s}{Bk} + 8r$$

sync operations is at most $\exp(-r)$.

Before giving the proof, let us summarize what the this theorem means specifically.

First, the condition $m + s \leq NB/5$ means that the guarantees hold only when at most 20% of the total backend capacity is utilized. For example, if using Dropbox with 1TB of available storage, the user should store at most 200GB of files in the frontend filesystem in order to guarantee the performance specified in Theorem 1.

Second, as mentioned already, the expected number of sync operations is optimal (up to constant factors), as the total amount of data written in the frontend cannot possibly exceed the amount of data being written to the backend.

In the number of syncs $16s/(Bk) + 8r$ required to clear the buffer with high probability, one can think of the parameter r as the number of "extra" sync operations required to be *very sure* that the buffer is cleared. In practice, r will be set proportionally to the security parameter. A benefit of our construction compared to many other ORAM schemes is that the performance degradation in terms of the security parameter is *additive* and not multiplicative. Put another way, if it takes 1 extra minute of syncing, after all operations are complete, in order to ensure high security, that extra minute is fixed regardless of how long the ObliviSync-RW has been running or how much total data has been written.

Finally, a key observation of this theorem is that it does *not* depend on the distribution of file sizes stored in the frontend filesystem, or their access patterns, but only the total size of data being stored. The performance guarantees of our system therefore allow arbitrary workloads by the user, provided they can tolerate a constant-factor increase in the backend storage size.

We now proceed with the proof of Theorem 1.

Proof. There are N blocks of backend storage. Each stores some combination of at most two split blocks and full blocks. Full blocks have size $\frac{B}{2}$ each, and split blocks contain multiple fragments summing to size at most $\frac{B}{2}$ each.

Suppose some sync operation occurs (selecting k block pairs from the backend, removing stale data and re-packing with new fragments from the buffer), and afterwards the buffer is still not empty. Then it *must* be that case that the k block pairs that were written are at least half filled, i.e., their total size is now at least $\frac{kB}{2}$. The reason is, if any block had size less than $\frac{B}{2}$, then it

could have fit something more (either a full block or a fragment) from the buffer. But since the buffer was not emptied, there were no entirely empty blocks among the k block pairs.

Furthermore, because $m < \frac{NB}{4}$ while the buffer is not empty, the *expected size* of a single, randomly-chosen pair of blocks is less than $\frac{B}{4}$. By linearity of expectation, the expected size of k randomly selected block pairs is less than $\frac{kB}{4}$.

Combining the conclusions from the preceding paragraphs we see that, on any sync operation that does not empty the buffer completely, the k randomly selected block pairs go from expected size less than $\frac{kB}{4}$, to guaranteed size greater than $\frac{kB}{2}$. This means the expected *decrease in buffer size* in each sync is at least $\frac{kB}{4}$. Starting with s bytes in the buffer, the expected number of syncs is therefore less than

$$\frac{s}{kB/4} = \frac{4s}{Bk}.$$

Now we extend this argument to get a tail bound on the probability that the buffer is not emptied after $T = 16s/(Bk) + 8r$ sync operations, for some $r \geq 0$.

Call a sync operation *productive* if it results in either the buffer being cleared entirely, or the size of the buffer decreasing by at least $\frac{kB}{4}$. From the discussion above, each sync will be productive with probability at least $\frac{1}{2}$.

Define a random variable X to be the number of lucky syncs among a series of $T = 16s/(Bk)+8r$ sync operations. Importantly, if $X \geq 4s/(Bk)$, then the buffer will be cleared at the end of T syncs.

We see that X is the sum of T i.i.d. Bernoulli trials, each with probability $p = \frac{1}{2}$. Therefore the Hoeffding bound from [13] tells us that, for any $\epsilon > 0$,

$$\Pr\left[X < \left(\tfrac{1}{2} - \epsilon\right) T\right] \leq \exp(-2\epsilon^2 T).$$

Setting $\epsilon = \frac{1}{4}$ works to bound the probability that $X < 4s/(Bk)$, since

$$\frac{4s}{Bk} < \frac{1}{4}\left(\frac{16s}{Bk} + 8r\right)$$

for any $r > 0$.

The theorem follows from the fact that

$$\exp(-2\epsilon^2 T) = \exp\left(-\tfrac{1}{8} \cdot \left(\tfrac{16s}{Bk} + 8r\right)\right) < \exp(-r).$$

■

5.2 Security

Theorem 2. *Consider* ObliviSync-RW *with parameters* B, N, k *as above, and with* drip time d. *For any* L *and* t *as fsequence parameters,* ObliviSync-RW *is strongly-secure write-only filesystem with running time* $T = t + \frac{16Ld}{Bk} + 8\lambda d$.

Proof. We need to show the following:

- ObliviSync-RW finishes all tasks in each sequence within time T with probability $1 - neg(\lambda)$.
- For any two (L, t)-fsequences P_0 and P_1, both access patterns are computationally indistinguishable from each other.

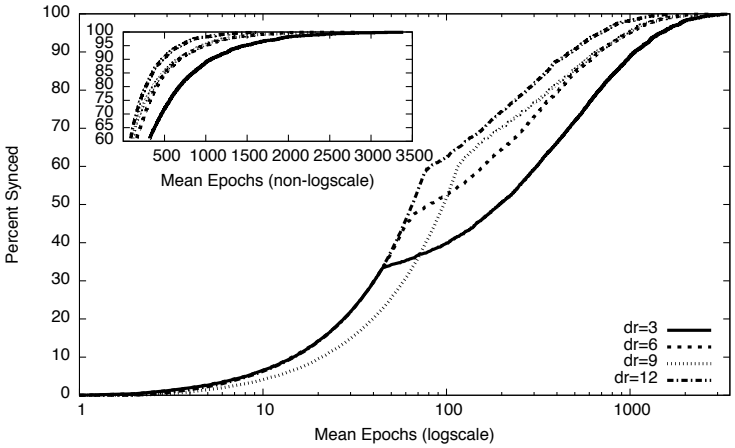

Figure 2: Throughput for Different Drip Rates (k)

The first condition is achieved according to Theorem 1, since one sync operation occurs every d seconds.

It is left to show that the second condition also holds. Obliviousness mostly follows from the original write-only ORAM security. To achieve strong obliviousness, we stress that ObliviSync-RW*always* writes encrypted data in k files at the backend chosen independently and uniformly at random. In particular:

- If there is too much data to be synchronized, the remaining data is safely stored in the temporary buffer so that the data will be eventually synchronized. Theorem 1 makes sure this must happen with overwhelming probability.
- If there is too little (or even no) data to be synchronized, the system generates what amounts to dummy traffic (re-packing the k chosen block pairs with the same data they stored before).

Therefore, the second condition is also satisfied. ■

6 Experiments

We fully implemented ObliviSync using python3 and `fusepy` [2], and the full source code of our implementation is available upon request. As explained in previous sections, our design can be naturally integrated with existing cloud services as Dropbox. For experimentation, where we are interested in measuring the throughput, latency, and buffer size, we cannot make precise measurements using third party software with proprietary implementations. Instead, we built in remote sync capabilities using `rysnc` that can synchronize the backends at regular intervals, say at the drip rate epoch. We found in our experiments that synchronization using `rsync` completed relatively consistently within two epochs of the ObliviSync, and hence adds just a constant latency to the overall system. We expect that commercial cloud synchronization services would have similar performance, which is orthogonal to the analysis of our construction. Based on this fact, we conducted further experiments outlined below using only local backend folders. That is, the ObliviSync-RO and ObliviSync-RW have a single backend folder that is shared between the two clients, meaning that changes propagate instantly.

Our remaining experiments focus on two main inflection points in the parameter settings of ObliviSync. First, we are concerned with the throughput and latency of the synchronization proce-

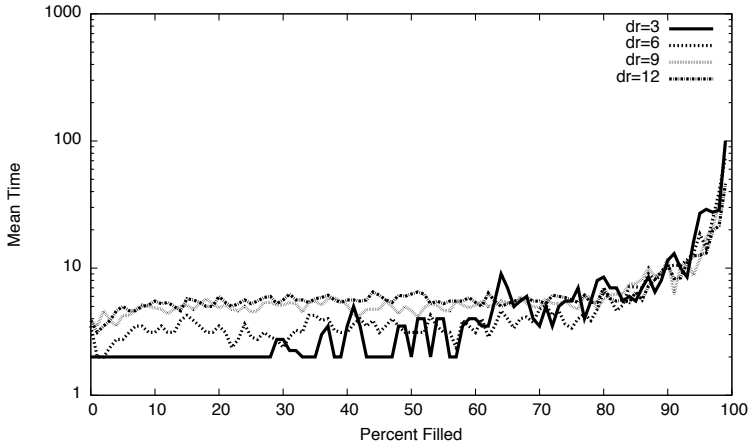

Figure 3: Latency for Different Drip Rates (k)

dure, seeking to understand how individual file updates propagate (latency) and how batches of file updates propagate (throughput). Second, we investigate the impact of the system with realistic file sizes based on published file system distributions [21, 9] at high thrashing rates and high storage loads.

6.1 Throughput vs. Latency

A key component of ObliviSync is how quickly data propagates from an ObliviSync-RW to an ObliviSync-RO, or the *sync rate*. One can consider two scenarios for understanding the sync rate with regard to the *throughput* of the system and the *latency* of the system.

(i) **Throughput**: If the user of ObliviSync were to insert a large number of files *all at once*, the buffer will immediately be filled to hold all the insertions. How how long does it take for each of the files to sync to the read end?

(ii) **Latency**: If the user of ObliviSync were to insert a large number of files *one at a time*, how long does it take for each of the files to sync to the read end?

We are keenly interested in the throughput and latency affects at the extremes, firstly when the blocks are mostly empty (below the 20% theoretic bound from Theorem 1), and when they are mostly full (much greater than the 20% bound). To limit the factors in these experiments, we used a backend with 2^{10} (1024) block pairs, and we attempted to insert $2^{10} - 2$ (1022) files each filling up two full blocks (including any metadata). The two remaining blocks must be reserved for the superblock and the directory entry. The blocksize was 2^{19} (512KB) so pairs of blocks are 1MB in size. The entire backend storage for the system was 2^{30} (1GB).

Throughput Measurement. In the throughput experiment, we established an empty ObliviSync-RW and attached an ObliviSync-RO to the backend. We then wrote 1022 two-block size files all at once to the ObliviSync-RW FUSE mounted frontend and monitored the ObliviSync-RO FUSE mounted frontend waiting for the new files to appear completely and correctly. In Figure 2, we graph the number of epochs (i.e., the number of timed sync operations) it takes for that percentage of files to synchronize.

Further, we conducted the experiments for different drip rates (k in the terminology of the previous section), the number of block pairs randomly selected at each epoch for writing. The

results presented are the average of four runs for drip rates set to 3, 6, 9, and 12. Note that there is an inherent trade-off between drip time and drip rate. Higher drip rates incur higher write costs and would mean that you would, in practice, need to increase the drip time (the length of an epoch) such that it does not take longer than then the drip time to perform the sync operation and read/write all the requisite data. For this reason, in our experiments, we also increased the drip time proportional to increasing the drip rate. (Otherwise, the higher drip rates would be strictly superior by simply writing more data within the same amount of time.)

Clearly, there are two regimes to this graph for all curves: linear and exponential synchronization. In the linear regime there are enough empty blocks that on each epoch progress is made clearing files from the buffer and writing new blocks to the backend. In the exponential regime, however, there are no longer enough empty blocks to reliably clear fragments from the buffer. Each additional block written further exacerbates the problem, so it takes an increasing amount of time to find the next empty block to clear the buffer further.

The inflection point, between linear and exponential, is particularly interesting. Apparent immediately is the fact that the inflection point is well beyond the 20% theoretic bound, even for a drip rate of 3, which goes exponential at about 33% (or 1/3 full). Further, notice that for higher drip rates, the inflection point occurs for higher percentage of fullness for the backend. This is to be expected; if you are selecting more blocks per epoch, you are more likely to find an empty block to clear the buffer. But we hasten to point out that there is a trade-off in practice here.

Latency Measurement. This trade-off between drip rate and drip time is made more apparent in the latency experiment; see Figure 3. Similar to the throughput experiment, we had ObliviSync-RW and ObliviSync-RO clients with a shared backend, writing to ObliviSync-RW FUSE mount and measuring synchronization rate to the ObliviSync-RO FUSE mount. To measure latency, we only add *one file at a time* and measured how long it took just that file to synchronize. As the number of files increases, filling the backend, the synchronization time increases. Again, we varied the drip rate, but for this experiment we set the drip time as small as possible for each epoch such that all requisite read/writes to the backend can complete within that epoch. The results are the average of four runs in each setting.

Again, there are two general regimes to the graph, and the transitions between them are, again, better than our theoretic 20% bound. First, for lower fill rates, the time to complete a single file synchronization is roughly two epochs. For example, for a drip rate of 3 we set the drip time to 1 second, and it takes two epochs to complete a single file synchronization. This is expected as it takes one epoch to write the file, and the second epoch to update the super block. However, higher drip rates requires higher drip times which decreases latency even for smaller fill rates. At higher fill rates, the advantage of lower drip rates fades, and overall, all drip rates perform equally poor. It should be noted, though, that even for very high fill rates, $> 90\%$, it only takes on the order of minutes to write a file, which for remote synchronization/backup scenarios may be sufficient for some applications.

Drip Rate vs. Drip Time. Based on these results, a clear question is then, *what is an appropriate drip rate and drip time setting?* The most important consideration is that the drip time must be long enough such that the writes to the backend can complete within the epoch before the next backend writes occur. If the system blows through this boundary, then privacy may be compromised, potentially revealing the size of the buffer — larger buffers mean more pending writes and more work for packing files into blocks. In practice, we found that a drip rate of 3 and drip time of 2 never experienced a boundary error for 1MB block pairs, but for Dropbox, which has 4MB blocks for files, a larger drip time may be appropriate.

This leads to the second consideration: the application. Synchronization rates of 2 seconds may

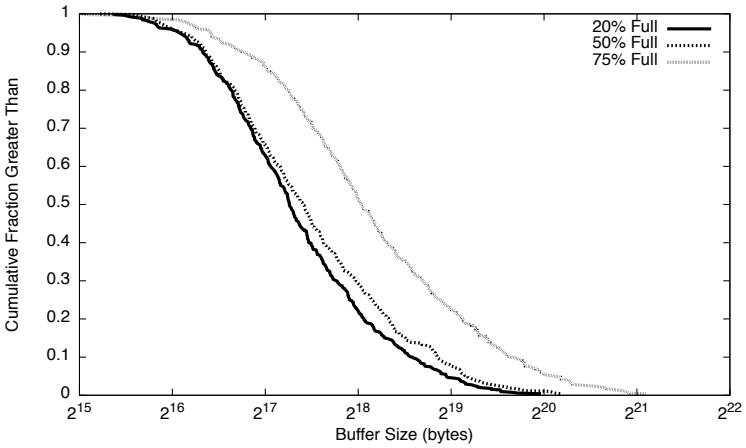

Figure 4: Buffer Size under Realistic File Distributions

not be appropriate for a daily backup system. It is quite reasonable to select a drip time on the order of hours which would allow for a much higher drip rate and thus much higher throughput, as would be preferred for a regular backup system. This trade-off actually highlights a benefit of ObliviSync's design that can enable high performance in multiple scenarios depending on the application domain and hardware/network limitations.

6.2 Realistic Workloads

In the previous experiments, the setup only considers fixed size files that fit cleanly into two full blocks each. In reality, files have variable size, and so we conducted further experiments with realistic file sizes under high thrashing rates.

To do this, we used the same 1GB backend with 1MB block pairs, but this time we inserted files of varied size based on known file size distributions such that the backend was filled to 20%, 50%, or 75%. The file sizes were based on prior published work in measuring file size distributions. In particular, we fit a lognormal distribution, which has been shown to closely match real file sizes [9], fit with data from a large-scale study of file sizes used in a university setting [21].

We then generate a series of writes to these files such that, on average, 1MB of data is updated on each write. This could occur within a single file or across multiple files. We selected which files to manipulate also based on the distributions of actual write operations in the same university study [21] that was used to generate the original file sizes. Roughly, this distribution gives a stronger preference to rewriting smaller files. We did not write *exactly* 1MB of data in each batch, but rather kept the average of each batch size as close to 1MB as possible in accordance with the experimental write size distribution. In particular, there were batches were a file larger than 1MB was written. In the experiment, we used a drip rate of 3 and drip time of 2 seconds, and (on average) 1MB of data was modified every second, half the drip time.

The primary result is presented in Figure 4 where we measure the number of bytes in the buffer on each synchronization. Clearly, as the fill rate increases, the amount of uncommitted data in the buffer increases; however, the relationship is not strictly linear. For example, with 20% full and 50% full, we see only a small difference in the buffer size for this extreme thrashing rate. The synchronization is able to keep up with the high thrashing rate for two main reasons: first, on each synchronization, it is generally able to clear something out the buffer; and second, some writes occur on the same files and on small files (as would be the case in a real file system), which

allows these writes to occur on cached copies in the buffer and the smaller files to packed together efficiently into blocks, even partially full ones.

At a fill rate of 75%, however, there is a noticeable performance degradation. Because most of the blocks selected at each epoch are either full or do not have enough space, due to fragmentation, the buffer cannot always be cleared at a rate sufficient to keep up with incoming writes. Thus, the size of the buffer doubles in comparisons with the other workloads.

7 Related Work

ORAM. ORAM protects the access pattern from an observer such that it is impossible to determine which operation is occurring, and on which item. The seminal work on the topic is by Goldreich and Ostrovsky [12], and since then, many works have focused on improving efficiency of ORAM in both the space, time, and communication cost complexities (for example [18, 15, 19, 14, 16] just to name a few; see the references therein).

Blass et al. introduced write-only ORAMs [8]. In a write-only ORAM, any two write accesses are indistinguishable, and they achieved a write-only ORAM with optimal $O(1)$ communication complexity and only poly-logarithmic user memory. Based on their work, we construct a write-only ORAM that additionally supports variable-size data and hides the when the data items are modified. We point out also that variable-sized blocks in traditional read/write ORAMs were also considered recently by [17], but with higher overhead than what can be achieved in the write-only setting.

Personal cloud storage. A personal cloud storage offers automatic backup, file synchronization, sharing and remote accessibility across a multitude of devices and operating systems. Among the popular personal cloud storages are Dropbox, Google Drive, Box, and One Drive.

However, privacy of cloud data is a growing concern, and to address this issue, many personal cloud services with better privacy appeared. Among the notable services are SpiderOak [5], Tresorit [6], Viivo [7], BoxCryptor [1], Sookas [4], PanBox [3], and OmniShare [20]. All the solutions achieve better privacy by encrypting the file data using encryption keys created by the client. We stress that however there has been no attempt to achieve the stronger privacy guarantee of obliviousness.

8 Conclusion

Contributions and Results. In this paper, we report our design, implementation, and evaluation of ObliviSync, which provides oblivious synchronization and backup for the cloud environment. Based on the key observation that for many cloud backup systems, such as Dropbox, only the writes to files are revealed to cloud provider while reads occur locally, we built upon write-only ORAM principals such that we can perform oblivious synchronization and backup with an approximate 4x overhead while also incorporating protection against timing channel attacks.

We also consider practicality and usability. ObliviSync is designed to seamlessly integrate with existing could services, by storing encrypted blocks in a normal directory as its backend. The backend can then be stored within any cloud based synchronization folder, such as a user's Dropbox folder. To be stored within the backend encrypted blocks, we designed a specialized block-based file system that can handle variable size files. The file system is presented to the user in a natural way via a frontend FUSE mount such that the user-facing interface is simply a folder, similar to other cloud synchronization services. Any modifications in the frontend FUSE mount are transparently

and automatically synchronized to the backend without leaking anything about the actual writes that have occurred.

In evaluating our system, we can prove that the performance guarantees hold when 20% of the capacity of the backend is used, and our experimental results find that, with realistic workloads, much higher capacities can in fact be tolerated while maintaining very reasonable efficiency. Importantly, ObliviSync can be tuned to the desired application based on modifying the drip rate and drip time to meet the application's latency and throughput needs.

Future Directions. There are a number of interesting future directions in this domain that are worth highlighting.

First, the current ObliviSync construction does not allow for the common practice of "de-duplication" where the cloud service can identify multiple users storing the same file across synchronization folders and then only store one copy of the file in the backend. The storage advantage that de-duplication provides is important to the economics of cloud service providers, and incorporating de-duplication (at least between trusted users) could be an interesting and powerful extension to ObliviSync.

While ObliviSync allows for multiple and concurrent reading clients, it does not allow for concurrent writers. Incorporating concurrent writing, either between the same user running multiple ObliviSync-RW clients, or among a group of trusted writers, would provide functionality that is more in line with current cloud service technologies. The existing buffer mechanism that stages writes for future backend commits may provide an insight into solving this problem for the write-only oblivious setting.

Finally, ObliviSync meets a stronger security definition than typical ORAM systems, in that any two operation sequences with the same total write size are indistinguishable. To do this requires fixing the synchronization parameters drip time and drip rate. However, we may be able to exploit a privacy/performance tradeoff by dynamically adjusting synchronization parameters to meet the changing demands of the client.

Acknowledgments

We thank Midshipman R. Blair Mason, whose undergraduate research project contributed to our design.

This work is supported in part by the National Science Foundation through awards #1406192 and #1319994, and by the Office of Naval Research through awards N0001416WX01489 and N0001415WX01532.

References

[1] Boxcrpytor. https://www.boxcryptor.com/en.

[2] fusepy. https://github.com/terencehonles/fusepy.

[3] Panbox. http://www.sirrix.de/content/pages/Panbox.htm.

[4] Sookasa. https://www.sookasa.com/.

[5] Spideroak. https://spideroak.com/.

[6] Tresorit. https://www.tresorit.com/.

[7] Viivo. https://www.viivo.com/.

[8] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.

[9] A. B. Downey. The structural cause of file size distributions. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*, pages 361–370, 2001.

[10] Idilio Drago, Marco Mellia, Maurizio M. Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: Understanding personal cloud storage services. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 481–494, New York, NY, USA, 2012. ACM.

[11] Inc Dropbox. Celebrating half a billion users, 2016.

[12] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[13] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.*, 58:13–30, 1963.

[14] Jonathan L. Dautrich Jr., Emil Stefanov, and Elaine Shi. Burst ORAM: minimizing ORAM response times for bursty access patterns. In *Proceedings of the 23rd USENIX Security Symposium*, pages 749–764, 2014.

[15] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.

[16] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication ORAM with small blocksize. In *ACM CCS 15*, pages 862–873. ACM Press, 2015.

[17] Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. A practical oblivious map data structure with secure deletion and history independence. In *2016 IEEE Symposium on Security and Privacy*, May 2016.

[18] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o ((logn) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214. Springer, 2011.

[19] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[20] Sandeep Tamrakar, Long Nguyen Hoang, Praveen Kumar Pendyala, Andrew Paverd, N. Asokan, and Ahmad-Reza Sadeghi. Omnishare: Securely accessing encrypted cloud storage from multiple authorized devices. *CoRR*, abs/1511.02119, 2015.

[21] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. File size distribution on UNIX systems: Then and now. *SIGOPS Oper. Syst. Rev.*, 40(1):100–104, January 2006.