Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward

Vincent Bindschaedler[1], Muhammad Naveed[1], Xiaorui Pan[2], XiaoFeng Wang[2], and Yan Huang[2]

[1]University of Illinois at Urbana-Champaign {bindsch2,naveed2}@illinois.edu

[2]Indiana University, Bloomington {xiaopan,xw7,yh33}@indiana.edu

www.oblivious-storage.com

ABSTRACT

To understand the gap between theory and practice for oblivious cloud storage, we experimentally evaluate four representative Oblivious RAM (ORAM) designs on Amazon S3. We replay realistic application traces to these ORAMs in order to understand whether they can meet the demands of various real applications using cloud storage as a backend. We find that metrics traditionally used in the ORAM literature, e.g., bandwidth overhead, fail to capture the practical needs of those applications. With a new understanding of the desirable properties, relevant metrics, and observations about the cloud services and their applications, we propose CURIOUS, a new modular partition-based ORAM framework, and show experimentally that it is thus far the most promising approach.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Measurement, Experimentation, Performance, Algorithms

Keywords

Oblivious Cloud Storage; Oblivious RAM; ORAM; Access Pattern

1. INTRODUCTION

Oblivious Random Access Memory (*ORAM*) is a security mechanism to hide data access patterns. This concept was proposed decades ago, for the purpose of hiding the way a program accesses memory to defend against software reverse-engineering [5]. Recently, the growing popularity of cloud computing has reinvigorated the research on the idea, which can potentially be applied to secure online storage systems that are increasingly being used to host personal or organizational data of critical importance (e.g., financial documents, health-care data, location information, etc.). Privacy concerns for those systems come not only from explicit data exposure, which could be addressed through encryption, but from possible side-channel leaks in which an unauthorized party (including the cloud provider) infers sensitive information from the pattern in which data is accessed. To mitigate this threat, numerous ORAM algorithms [16, 9, 25, 24, 22] have been proposed, particularly to make this oblivious access technique more applicable to the cloud.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813649.

Rethinking ORAM on the cloud. Despite significant progress made on ORAM designs, when it comes to the cloud application, it is still unclear whether this theoretic concept is indeed moving towards the practical end. A *fundamental* question one may ask is: *Whether the security model and performance metrics of today's ORAM designs are in line with the constraints and demands of real-world cloud services and applications?* As mentioned earlier, the concept of ORAM originates from memory protection, which does not take into account unique properties of the modern cloud. Examples include its *elastic resource allocation* and *asynchronously running applications*, which all affect the security assurance by the ORAM primitive (Section 4.2, [20], and [13]). On the performance front, conventional ORAM algorithms seek to minimize the client-side storage (e.g., logarithmic of the outsourced storage, or even constant [19]), and the data-access bandwidth. This is increasingly out of the touch with the reality of modern computing. Indeed, storage has become increasingly cheaper, and even a smartphone today can easily afford gigabytes of space for supporting security-critical services. It is conceivable that an (otherwise-preferable) ORAM scheme requiring linear local storage works well in practice, as long as the *outsourcing ratio* is reasonably high (e.g., 1 TB on the cloud vs. 1 GB on a mobile device). Given such misalignments, together with the technical challenges in making ORAM efficient, we may further ask: *How close the current ORAM techniques are to offering any practical support for cloud applications? Are we on the right track to narrow this gap?*

These questions can only be answered by a systematic study of the operations of state-of-the-art ORAM schemes over real-world cloud storage systems. This is a challenging

task. First, most existing ORAM approaches do not have prototypes suitable for running on the cloud. Realizing theoretical ORAM constructions on existing cloud storage platform requires additional careful examination and engineering efforts. Take Path ORAM [22] as an example to see the gap, although the full protocol can be described by merely 16 pseudo instructions, a single instruction that reads data blocks from a path needs tens of lines of Java code for carefully issuing asynchronous requests to the cloud. Second, running a real cloud application on top of ORAM in practice requires changes to all its storage access instructions (see Section 3.1), which is complicated due to the mismatch in the storage API designs and the low availability of the source code of existing ORAM implementations. Actually, never before has *any* ORAM implementation been tested on a commercial cloud storage system, not to mention any attempt to understand its effectiveness in supporting real-world cloud applications.

Understanding the gap. Understanding the gap between the idea of ORAM and *today's cloud reality* is the only way to keep the research on the right track. Since it is unlikely that the cloud architecture, implementation and business model will be changed to suit the ORAM designs, we argue it is important to adjust ORAM designs to better fit it with the cloud. To this end, we built cloud-capable versions of four most representative ORAM algorithms, including a layered ORAM [7], a tree-based ORAM (PathORAM [22]), a partition-based ORAM (ObliviStore [20]) and a large-message ORAM [8][1]. These implementations were evaluated against realistic cloud applications and workloads using a unique methodology. More specifically, we connected *Filebench* [18], a popular storage benchmark, to Amazon S3, and ran the tool to produce a large amount of diverse storage-access workloads. Such workloads realistically describe the real-world operations of a variety of cloud-related services and applications, e.g., web servers, file servers, etc. Their traffic traces were captured on the cloud end and then replayed to our ORAM implementations, which were also connected to S3.

We found that today's ORAM designs are still far from practical deployment. For example, for some task, even the most efficient design, ObliviStore, took 181 seconds to handle what was accomplished within 38.5 seconds in the absence of the ORAM, generating more than 17 times of additional traffic and costing its user 68 times more in terms of monetary expense. For the most optimistic task they could handle, other approaches incurred between 3.3 and 32.8 times of performance slowdown and between 53 and 795 times extra fees for renting the Amazon service. Most importantly, our research brought to light significant and surprising misalignments between the metrics used to evaluate an ORAM scheme and those to evaluate a cloud application. Traditional metrics for evaluating ORAM (e.g., bandwidth) do not fully reflect its ability to support cloud applications. The mismatch can be explained with several overlooked features of cloud services and overlooked application requirements. For instance, compare uploading with downloading the same amount of data, uploading is much slower (1.5 to 1.8 times for S3) than downloading; and it also costs more money (12.5 times for S3). In addition, data access patterns can also affect the performance and monetary cost in substantial ways. Missing these points, as do all existing approaches, a conventionally appealing ORAM construction may turn out to be unusable in cloud environment. Further, many ORAM designs (e.g., layered and tree-based family of ORAMs) assume a sequential requests model, which makes them ill-suited for a majority of modern cloud applications that would be unusable without concurrent data access (Section 4.2). Additionally, we found that some applications cannot be supported by synchronous ORAM schemes (existing or future), because they also need to manipulate variable-size objects (recall all existing ORAMs assume fixed-size blocks). Last, the eviction process of many popular ORAMs are incompatible with some applications such as back-up services because certain fresh data is cached on the client side and if the client crashes, it gets lost.

Although some of these missing design metrics have been mentioned by some prior studies (such as elasticity in [13] and asynchronicity in [20]), our research shows that adding them to the ORAM design is more difficult than thought: for example, asynchronous operations have not been done right in any existing implementation. We found the prototype of ObliviStore leaking information in the presence of multiple requests for the same data block (Section 4.2). Further, these properties do not appear to be the primary focus of the mainstream ORAM research. After inspecting 39 research papers on this subject and the citations they receive (Section 5.2), we found that most recent work continues to focus on the constructions that are less likely to support cloud applications. This finding suggests that the research in this area could actually be moving away from one of its major objectives, i.e., cloud storage protection.

Seeking new solutions. Based upon this new understanding, we identified a new set of metrics for evaluating ORAM designs, focusing more on latency, monetary expense, outsourcing ratio, elasticity, and reliability. Among all existing designs, ObliviStore, which is built on partitioning the main ORAM into a set of smaller server-side ORAMs, turns out to be the most promising one. However, we found that in addition to its privacy weakness in its implementation, the construction is overly complicated due to some of its specific performance optimization (e.g., background shuffling). In our research, we come up with a new ORAM design, called *CURIOUS* (Cloud Framework for Oblivious Storage). CURIOUS is characterized by a set of *fixed-size small ORAMs*, offering a large constant outsourcing ratio, convenience for supporting asynchronous operations and the capability to expand and shrink its cloud-side storage. It has been carefully built to ensure oblivious data access when serving multiple requests concurrently, and adopt a simpler eviction strategy, making it easier to implement. Also importantly, unlike ObliviStore, which is tied to a layered ORAM scheme [20, 21], CURIOUS allows its underlying small, fixed-size ORAMs to be easily replaced. As a result, its performance will be continuously improved whenever a new design of such a building-block ORAM is available. For applications easily supported by ORAM, both ObliviStore and CURIOUS perform comparably. However, for demanding applications that stressed ORAM, CURIOUS significantly outperforms ObliviStore in response time (only its 25%), despite doubling the network traffic. In all cases, CURIOUS incurred lower monetary expense than (1/2 $\sim$ 2/3

[1]These systems were all developed by strictly following their publicly available descriptions and leveraging existing code when possible (Section 3.2).

of) ObliviStore. Our design serves as a starting point for further efforts to bring ORAM closer to its deployment for cloud storage.

Contributions. Our contributions include:

• *New understanding.* We made the first attempt to evaluate mainstream ORAM designs on the real-world cloud storage and applications. Our study reveals not only the performance gap between the state-of-the-art of ORAM research and the cloud reality, but also some fundamental design requirements missing in today's ORAM constructions. Such miscalibrations risk moving the area further away from practice, as suggested by the trend of the proposed new constructions (Section 5.2). Better understanding of what needs to be done and what should be avoided is critical for putting the research back on the right track.

• *New technique.* We developed CURIOUS, a modularized framework for supporting oblivious storage access on the cloud. Compared with prior approaches, CURIOUS is a step closer to supporting real-world applications with oblivious cloud storage. To encourage further research, we make the code of the CURIOUS framework available at `http://oblivious-storage.com`.

2. BACKGROUND AND PRELIMINARIES

2.1 Cloud Storage

A prominent example of online cloud storage is Amazon Simple Storage Service (S3). S3 allows customers to organize their data into *buckets*, each of which has a separate name space, holding a large number of objects such as files. An S3 user can perform five bucket operations through the Amazon APIs, including `get` or `put` that downloads and uploads objects respectively, and `list`, `copy`, or `delete` that performs the functionalities of the UNIX commands `ls`, `cp` and `rm` respectively. The service provider (Amazon) determines the user's bill based on the number and type of operations she performed (e.g., 0.004 USD per 10K `get`s, 0.005 USD per 1000 `put`s, `list`s, and `copy`s), the amount of network traffic (e.g., 0.09 USD per GB for up to 10 TB/month), and the storage used (e.g., 0.03 USD per GB for the first 1 TB/month). Other popular cloud storage systems, like Google Cloud, all have similar APIs and pricing structures.

Throughout the paper, we consider a typical way of using the cloud storage system: an application running on the client device (e.g., the user's in-house server, desktop, smartphone, etc.) integrates the *storage interface* (i.e., the aforementioned APIs) and utilizes the interface to operate on its cloud-side data. For example, a mobile app could use the interface to upload the user's pictures onto the cloud. Note that we do not consider the setting where both the application and its storage sit on the same cloud, as used in the prior research [10].

2.2 Oblivious RAM

An Oblivious RAM scheme is a trusted mechanism on a client, which helps an application or the user access the untrusted cloud storage. For each read or write operation the user wants to perform on her cloud-side data, the mechanism converts it into a sequence of operations executed by the storage server. The design of the ORAM ensures that for any two sequences of requests (of the same length), the distributions of the resulting sequences of operations are indistinguishable to the cloud storage. Existing ORAM schemes typically fall into one of the following categories: (1) layered (also called hierarchical) [5, 25], (2) partition-based [20], (3) tree-based [19, 22, 2, 23, 11, 17]; and (4) large-message ORAMs [1, 8, 15].

In this paper, we consider a *block* to be the unit of ORAM data that is uniquely identified by its *key* (i.e., a block identifier). To access a block with key x, an application sends a request to the ORAM mechanism running on the client side. The ORAM client then performs the operations through issuing a sequence of cloud storage instructions.

Representative schemes. A layered ORAM organizes n data blocks on the storage into a hierarchical structure with $\log n + 1$ levels. Each level consists of a hash table, whose size increases exponentially: the size of the table on layer i doubles that on layer $i-1$. Typically, Cuckoo hashing [16, 9] is used to realize the hash tables. To access block x, the ORAM client proceeds layer by layer (from the lowest layer upwards) reading all blocks using the hashes of x as indices (except for layers after x is found, where random dummy blocks will be retrieved). Once the query is done, block x is stored back into the hierarchy (in the lowest layer with an empty entry). Periodically, all blocks in layers below and including layer i will be reshuffled and pushed into layer $i+1$.

A tree-based ORAM organizes the external storage as a tree where each tree node holds a few blocks. To serve a request to access x, the ORAM client looks up a position map that links a block to a tree leaf. From the leaf, all blocks on the path to the root is then retrieved to find the block x. An eviction procedure follows that aims to save x somewhere back in the tree. A representative tree-based ORAM is PathORAM [22].

A partition-based ORAM divides its external storage into m partitions, each managed separately. The client maintains a position map to keep track of each block's partition, and an eviction cache to hold the block downloaded from the storage for a random amount of time, before re-writing it back to a random partition. So far, only two (very similar) partition-based schemes have been proposed, which are all based upon simplified layered ORAM (without Cuckoo tables). A prominent example is ObliviStore [20].

A large-message scheme [8, 1, 15] maintains a cache on the client side. The ORAM simply reads from the cache when the requested block is there, or fetches it from the external storage and then puts the block in the cache afterwards. Once the cache is full, the entire external storage has to be rebuilt through an oblivious sorting algorithm.

We do not consider schemes which require server computation such as [24, 3, 11, 4, 14], since they are not supported in our setting.

3. ANALYSIS OF ORAM ON THE CLOUD

3.1 Methodology

At a high level, our methodology is to run a set of popular applications that utilize cloud storage (e.g., web servers, email servers, web proxies, etc.) on the client system (in-house server, workstation, mobile devices etc.), under realistic workloads, to access their cloud-side data through an ORAM client. The idea is to understand how the operations for achieving obliviousness affect the performance of the application, whether such impacts are in line with

what is predicted by existing metrics, and also what privacy implications the ORAM design could have in a real cloud-computing environment. It is important to note that we do not consider the scenarios where cloud-side supports are provided to facilitate ORAM operations, as assumed in some prior research [11, 24], since it is less clear whether cloud service providers are willing to change their service model to support ORAMs and even if so, how they are going to do that. Without support from actual providers, it is hard to conduct a preliminary study in that direction.

When it comes to the details, two issues must be addressed to make the methodology work: we need to select from tens of ORAM designs the most representative and promising ones for evaluation, and also find a way to incorporate them into the applications using cloud storages.

Choosing ORAM representatives. As mentioned earlier, mainstream ORAM designs can be roughly classified into four categories: layered ORAMs, tree-based ORAMs, partition-based ORAMs, and large-message ORAMs. We selected a representative from each category. Specifically, for layered ORAMs, we picked the asymptotically most efficient de-amortized worst-case layered scheme [7] (which we call *LayeredORAM*) that utilizes Cuckoo hashing and Randomized Shell Sort. For tree-based approaches, we chose PathORAM [22], a construction known for its efficiency and simplicity. There are only two partition-based designs and ObliviStore [20] is the one that was evaluated on Amazon EC2 (not S3) and achieved a high throughput. In the large-message ORAM category, we selected *PracticalOS* [8], due to its improved performance over previous schemes in the same category such as [1]. None of these have prototypes suitable for running on cloud storage, so we implemented them (Section 3.2).

We did our best to select the most representative ORAM schemes, typically the most efficient ones (asymptotically) in their respective categories. We believe the results of our study at least offer new insights about how the state-of-the-art ORAMs could work in a real-world cloud storage system. We emphasize that our goal is to understand the gap between current designs and what cloud applications require; not merely identify the best ORAM scheme.

Evaluating ORAM supports for cloud apps. To understand how well these ORAM designs are up to the task of supporting cloud applications, we need to evaluate applications on top of the ORAM implementations. As discussed earlier, this is nontrivial due to the challenges in integrating ORAM interfaces into those applications and further generating realistic workloads during their runtime. In our research, we adopted a unique approach, which recorded the communication traces between an application workload generator and its cloud-side storage, and then replayed the traces to our ORAM implementations.

Specifically, we used *Filebench* [18, 12], a popular performance benchmark capable of emulating the workloads of many complex applications (e.g., mail, web, file and database servers), as the workload generator. In our research, we mounted an S3 folder (i.e., a bucket) to a client system using S3FS and ran Filebench over this folder. After executing a task, e.g., simulating a web server's workload for ten minutes, we retrieved from S3 a server access log file recording request details (including the type of operations (`get`, `put`, etc.), the arrival time of each request, the number of bytes delivered etc.) Amazon received. An example trace is illustrated in Figure 1.

```
1424659600, GET, bigfileset/00000001/00000145, 51738, 51738
1424659600, GET, bigfileset/00000001/00000140, 31558, 31558
1424659600, GET, bigfileset/00000001/00000143, 3621, 3621
1424659600, PUT, bigfileset/00000001/00000044, -, 227
1424659600, PUT, bigfileset/00000001/00000142, -, 49297
1424659600, PUT, bigfileset/00000001/00000042, -, 201
1424659601, PUT, bigfileset/00000001/00000145, -, 64387
1424659601, PUT, bigfileset/00000001/00000046, -, 6808
1424659601, GET, bigfileset/00000001/00000148, 17139, 17139
```

Figure 1: Excerpt trace of the varmail filebench application. It contains the following fields (in order): timestamp (UNIX format), type of request, name of object/file requested, number of bytes read (if `get`), total size (in bytes) of the object/file.

To evaluate an ORAM, we then replayed the traces to the ORAM client that communicated across the Internet with S3. The ORAM client kept track of the response time for each request (which may contain multiple round-trips between the client and S3), the amount of data exchanged, monetary cost, and other performance data. We further compared these measurements with a baseline client that did not use ORAM (nor providing access pattern privacy).

Adversary model. We make the standard assumptions made by other ORAM research: the cloud is an honest-but-curious party; but, the client is trusted.

3.2 Optimistic Implementation

Surprisingly, we found that building those ORAM approaches was complicated by the lack of design details — the papers [7, 20, 21] did not describe many important design details for developing working systems. To fill up the gap, we contacted the authors and acquired the prototypes of ObliviStore and PathORAM, which we modified to make them work on a cloud storage system. However, since the implementation of LayeredORAM and PracticalOS was not available, we adopted an "optimistic" way to implement their schemes. PracticalOS could be implemented fully from the description. However, for LayeredORAM, we just built the parts explicated by the paper and simplified the components whose details are missing there (see below), so it did not fully function as an ORAM, yet in terms of performance, it should outperform the fully implemented version, as only part of the overall overhead is actually measured.

LayeredORAM [7]. The paper of LayeredORAM is quite detailed about the way the worst case performance is achieved and the ideas behind de-amortizing the scheme. This enabled a faithful implementation in our cloud storage experiments. However, when it comes to the concurrent Cuckoo table rebuilding, which includes concurrent reshuffling and the concurrent oblivious Cuckoo hash tables construction, we found that these details were missing. Specifically, it gave reference pointers on randomized Shell-sort [6] and the oblivious hash table rebuilding, but did not explain how those operations can be concurrently spread out over multiple requests (which is needed for de-amortization). The paper only specifies that $2b$ accesses have to be made "towards a rebuild" (of each layer's Cuckoo table) for some $b > 0$, without specifying what those accesses are. As a result, we could only construct a component that "emulated" what was expected to be done. More specifically, we performed $2b$ dummy accesses, i.e., b (concurrent) `get`s fol-

lowed by b (concurrent) `puts`. Because we optimistically set $b = 1$, our implementation can only lead to underestimation of the scheme's cost. Also, the paper does not give concrete values for some critical parameters, including epoch size $q = O(\log n)$, and stash size $2s \log n$ for some $s > 1$. Optimistically, we set $q = \log_2 n$, and $s = 2$.

PathORAM [22]. Conceptually, PathORAM is rather simple; it can be described by merely 16 lines of pseudo-code. The implementation, however, is more complicated. Even with the prototype we obtained, we still made significant efforts to make it work with S3. As an example, in the PathORAM paper [22], retrieving a path was described in a single line of code. When performing this operation, however, the ORAM client first needs to calculate the path locally upon the data structure describing the tree, then schedule the queries (for different blocks on the paths on the S3) asynchronously and wait for their completion. In the end, the prototype we used for our study includes over 1000 lines of code. Based on the most optimistic parameter setting in the original paper [22, Table 3], we chose the number of blocks per tree-node $z = 4$ and stash size $s = 89$.

ObliviStore [20]. Among all the schemes we tested, ObliviStore is the most complex one. The complexity stems from its attempts to support asynchronous request processing and backgrounded shuffling and eviction. This calls for subtle synchronization to avoid messing up the system's states (e.g., the shuffling should not be executed before all the blocks have been read). Unfortunately, such critical details on how to concurrently perform those operations efficiently and securely was missing in the paper. The prototype provided by the authors utilizes two semaphores to keep some level of synchronization between shuffling, requests processing, and local memory operation. It runs a single thread to sequentially schedule these tasks: each request or background shuffling task is put in a single queue and executed one by one; this ensures that there are no concurrent modifications of the local state that would result in inconsistency. Note that the tasks being scheduled can still run concurrently, in the sense that the system does not need to wait for the completion of one task before invoking another. The complexity in asynchronous and concurrent processing actually makes it more difficult to achieve obliviousness, as we discuss in Section 4.2.

The original prototype of ObliviStore was in C#, while PathORAM and other ORAM prototypes were built in Java. To compare these systems in the same environment (which was convenient for the experiment setting and data collection), we re-implemented ObliviStore in Java by strictly following the code provided by the authors, and just adding the functionalities to interact with S3. Note that the performance impact of the programming languages is minimum here, because all the delays we observed in our study were predominantly caused by communication and cloud-side operations. More specifically, we observed that the client-side processing (e.g., scheduling a request) is on the order of 1ms or less, whereas the processing time of those requests, i.e., interaction with S3, is on the order of 100ms or more.

PracticalOS [8]. We fully implemented PracticalOS. The only issue we encountered was that the authors made assumptions about a few cloud APIs that S3 and other cloud-storage services (like Google) do not actually provide. For example, `getRange` was assumed in the original design to retrieve a set of blocks with consecutive keys in one single operation. To address this inconsistency, we had to utilize the existing storage interfaces to "emulate" those APIs. Specifically, `getRange` was realized with `list` and a number of `get` operations performed in parallel, as suggested in [8, Section 7]. We further enabled sending requests to the cloud storage asynchronously, for both `get` and `delete` operations. However, the re-shuffling process, cannot be performed concurrently with outstanding requests, because all data blocks, including those in the local dictionary [8], are required to be obliviously sorted together. The paper provides constructions for arbitrary $c \geq 2$, where c indicates the amount of local memory being used. We chose $c = 2$, which uses the most memory, but has the lowest communication overhead.

4. FINDINGS

Experiment settings. Our experiments were conducted from a Linux server on a university network. The machine ran an ORAM client to interact with S3; the S3 buckets were placed on the US_EAST1 Standard (North Virginia) Amazon S3 region. This region has the lowest round-trip time with the ORAM client. The bandwidth between the client and S3 was 50 MB/s downstream and 10 MB/s upstream[2]. In our experiments, we always started running an ORAM in a warmed-up state (after $O(n)$ requests were processed where n is the capacity of the ORAM instance).

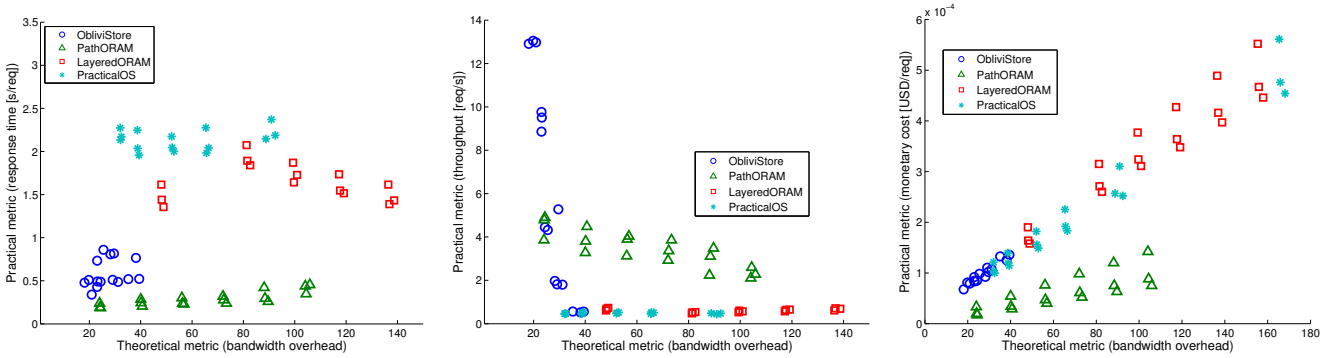
4.1 Results: the Landscape

Overheads. We ran experiments to find out the per-block access overhead for each of the four ORAM schemes. For this purpose, we generated a trace of single-block requests, under different ORAM capacities — 1 MB (256 × 4KB blocks) versus 1 GB (2^{18} × 4KB blocks) — and replayed it, one request at a time, to each ORAM implementation. The trace was also replayed directly to S3 without going through an ORAM. In each case, 3 sets of 100 random requests (for random blocks) were replayed and the results are presented in Table 1. We measured: the bandwidth usage (i.e., KB downloaded and uploaded per request), mean response time (i.e., time from when the request is scheduled to when it is completed), and outsource ratio (i.e., ratio between the amount of data stored on the cloud and locally)[3]. Comparing to the baseline, all tested ORAM schemes incurred significant overheads. While ObliviStore stands out with the lowest bandwidth usage and response time, its response time is nearly doubled when its capacity goes up from 1 MB to 1 GB (i.e., 0.156s vs. 0.079s). In contrast, the response time

	Scheme	Without	ObliviStore	PathORAM	LayeredORAM	PracticalOS
1 MB	Bandwidth	4	79.26	287.95	468.48	660.68
	Resp. time	0.076	0.079	0.322	1.544	2.386
	Outsource	∞	1.57	2.87	N/A[4]	7.96
1 GB	Bandwidth	4	140.06	608.05	846.64	-
	Resp. time	0.085	0.156	0.406	1.487	-
	Outsource	∞	538.50	929.35	N/A	-

Table 1: Overhead of single-block requests. Bandwidth usage (in KB/req), mean response time (sec), and outsource ratio were measured, for ORAM capacities: 1MB, 1GB; the block size was 4KB.

[2] Measured through the download/upload of a 100MB file from/to S3.
[3] In all experiments, we calculate the outsource ratio experimentally based on the *peak* memory usage of each scheme during the replay.



(a) Response time (b) Throughput (c) Monetary cost

Figure 2: Theoretical metric (i.e., bandwidth overhead) against a variety practical metrics for the considered ORAMs. For all schemes we consider block sizes of 1KB, 4KB, and 16KB. The considered number of blocks (i.e., n) were: 4, 16, 64, 256, 1024, 4096, for PathORAM and LayeredORAM; 1024, 4096, 8192, 16384, 65536, 262144, for ObliviStore; and 4, 8, 16, 32, 64, 256, for PracticalOS.

of PathORAM for 1GB is only 26% higher than for 1MB. This is surprising because in both cases the bandwidth usage for 1GB is roughly twice that of 1MB, while bandwidth was one of the foremost performance metrics. Yet, it did not seem to correlate well to the response time across schemes.

To better understand this phenomena, we ran more experiments with a wider range of ORAM capacities and block sizes. Besides response time, we also measured throughput (in number of requests per second) and monetary cost. We varied the number of blocks to keep the bandwidth consumption in different schemes within a comparable range. Figure 2 plots the relations between bandwidth as a metric and several practical metrics such as response time, throughput, and monetary cost.

We observed that, when bandwidth was not saturated, bandwidth as a metric did not reflect very consistent effect on other metrics such as response time, throughput and monetary expenses. This observation was found to be true both within and across schemes. For example, PathORAM, which consumed more bandwidth than ObliviStore, sometimes resulted in lower response time. Even though monetary expense exhibit a strong linear correlation to bandwidth overhead, not all schemes follow the same line; also PathORAM resulted in lower monetary cost than others despite its higher bandwidth usage. This indicates a mismatch between ORAM's theoretical design and its the practical performance considered in a cloud environment.

Misalignments. To further explore this misalignment, we measured the performance of the ORAM schemes in the context of different applications. Specifically, we replayed various Filebench traces with and without ORAM[5], with a varying number of application-level requests (i.e., 200 for varmail, 500 for webproxy, and 1000 for both webserver and fileserver)[6]. The replay was done according to the actual timings of requests produced by different applications as recorded in the trace.

A real-world cloud application needs to handle multiple service requests, while the service time experienced by the user is also affected by other requests. To understand whether current ORAM schemes could support such applications, we further measured the *slowdown* factor on batches of requests, i.e., the ratio between the delays to complete all requests in a batch, with and without ORAM.

During an oblivious data access, each *application-level request* (e.g., getting an HTML page) needs to be translated into one or more *ORAM requests* (i.e, read/write blocks), and the ORAM request further involves multiple rounds of interaction with the cloud storage server. When an object cannot be accommodated by a single block, an application-level request needs to be fulfilled by a sequence of ORAM requests. This is why the response time when replaying application-level requests (Table 2) is longer than when replaying individual ORAM requests (Table 1). In general, a request initiated by the application will only be served after all related blocks have been obliviously retrieved from (for a `get`), or written to (for a `put`) the cloud storage. The size of ORAM blocks also affects ORAM performance. In our experiments, we tested two sizes for each application, 16 KB and 64 KB for fileserver, and 4 KB and 16 KB for others (varmail, webserver and webproxy). Those are also the typical parameters adopted by prior ORAM schemes [20].

The outcomes of our study are illustrated in Tables 2 and 3. In addition to the four ORAM schemes tested, we include two baselines: (1) "No ORAM" and (2) "No ORAM with Blocks", in which files were split into fixed-size blocks.

The results indicate that except ObliviStore, none of the other ORAM schemes handles the applications well, as evidenced by their substantial slowdowns. ObliviStore was found to work well with both varmail and webproxy, but it was unable to keep up with webserver and fileserver, which were more intensive (sending requests more frequently) than other applications. In terms of the monetary cost, all these schemes cost significantly more than the baselines. For example, running those applications on top of ObliviStore incurred 25 to 200 times the expense of not using ORAM ObliviStore's bandwidth overhead ranges from 17.24 to 40.66 per request, and it's outsource ratio varies from 14.9 to 124. Similar to the results in Figure 2, we find that the bandwidth overhead is not strongly correlated to other practical metrics. For example, for fileserver with block size 16KB, the bandwidth overhead is at its lowest (17.24) among all combinations of trace and block size, while the monetary cost overhead is $0.546/0.008 = 68.25$ (more than 2.5 times its

[4]For reasons described in Section 3, we are unable to calculate the outsource ratio of LayeredORAM.

[5]In the experiment, we set the ORAM capacity to an appropriate value, since each trace manipulates a different amount of data.

[6]We varied the number of requests based on the intensity of the communication. For example, varmail has lower intensity than other traces: 200 requests is roughly 40 seconds of real time.

		Application							
		varmail (64 MB)		webserver (256 MB)		webproxy (256 MB)		fileserver (256 MB)	
Scheme	Metric	4 KB	16 KB	4 KB	16 KB	4 KB	16 KB	16 KB	64 KB
Baseline (No ORAM)	Bandwidth use (KB/req)	15.96		15.90		14.74		120.04	
	Response time (sec)	0.108		0.093		0.095		0.201	
	Cost (USD/1000 reqs)	0.004		0.002		0.002		0.008	
No ORAM (blocks)	Bandwidth overhead	1.06	1.24	1.00	1.00	1.03	1.09	1.05	1.16
	Response time (sec)	0.125	0.122	0.150	0.094	0.111	0.102	0.186	0.172
	Relative slowdown	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Cost (USD/1000 reqs)	0.013	0.005	0.003	0.002	0.005	0.003	0.028	0.012
ObliviStore	Bandwidth overhead	28.27	35.69	24.92	33.74	30.81	40.66	17.24	20.81
	Response time (sec)	0.546	0.169	10.895	3.613	2.454	0.184	15.323	5.105
	Relative slowdown	1.002	1.000	3.439	1.246	1.022	1.000	4.716	1.318
	Outsource Ratio	120.6	80.8	35.9	30.7	124.0	103.4	14.9	15.1
	Cost (USD/1000 reqs)	0.412	0.142	0.356	0.132	0.405	0.146	0.546	0.205
PathORAM	Bandwidth overhead	135.78	157.65	153.98	185.84	155.67	189.61	128.10	132.58
	Response time (sec)	1.564	0.605	1.628	0.726	1.633	0.630	3.487	1.977
	Relative slowdown	8.533	3.303	71.473	31.858	8.849	3.476	91.779	52.025
	Outsource Ratio	168.6	45.7	540.0	180.0	540.0	180.0	180.0	45.9
	Cost (USD/1000 reqs)	0.458	0.214	0.518	0.251	0.486	0.238	1.309	0.851

Table 2: Comparison of ORAM schemes replaying various applications traces (see Section 3.1), for various block sizes, and according to various metrics: mean response time per request (in seconds), relative slowdown, outsource ratio, and monetary cost (in USD per 1000 reqs).

		varmail (64 MB)	
Scheme	Metric	4 KB	16 KB
LayeredORAM	Bandwidth overhead	230.28	258.25
	Response time (sec)	11.641	3.491
	Relative slowdown	63.7	19.1
	Cost (USD/1000 reqs)	2.747	0.909
PracticalOS	Bandwidth overhead	1394.12	974.13
	Response time (sec)	22.824	6.084
	Relative slowdown	124.1	32.8
	Outsource Ratio	63.7	32.0
	Cost (USD/1000 reqs)	15.436	3.180

Table 3: Replay of varmail with LayeredORAM and PracticalOS, for block sizes 4KB and 16KB, respectively.

lowest value). Also, PathORAM uses a significant amount of bandwidth, the overhead ranges from 128.10 to 189.61, which is 4 times or more than that of ObliviStore. Yet its operating monetary cost in most cases comes below two times that of ObliviStore. Moreover, though PathORAM offers a higher outsource ratio, its slowdown is significant in all cases.

Table 3 shows the performance metric for both LayeredORAM and PracticalOS for the varmail application. Both schemes simply cannot handle this application. For PracticalOS, the median response we measured is quite low (e.g., 0.147 for 4KB blocks, and 0.117 for 16KB blocks) but the mean response time (Table 3) is high due to the expensive reshuffling operation (e.g., 6.084 for 16KB block size). That is, in terms of response time without considering reshuffling, the scheme is very competitive. However, even if the reshuffling is performed "offline" (e.g., during periods of inactivity, thus does not directly impact application performance), the scheme is still prohibitively expensive monetarily (costing more than 15 USD per 1000 application-level requests for 4KB block size). Due to their inefficiency (e.g., it took PracticalOS more than one hour to replay a trace running 35 seconds on baseline, for 4KB block size.) and high monetary cost, we did not consider further experiments for LayeredORAM and PracticalOS.

4.2 Understanding the Cost

To gain insights into the aforementioned findings, we looked at the specific features of each ORAM scheme to understand: (1) what is causing the discrepancy between the theoretical metric and the practical ones, and (2) what allows some schemes to handle real cloud applications, while other schemes cannot. To this end, we compared different schemes, focusing on their features that could make a difference in the throughput, latency, and monetary expense they could get. We made the following discoveries.

Asynchronicity. An interesting observation is that PathORAM performed reasonably when dealing with single block requests but caused a huge slowdown when processing application requests in batch (e.g., 71 times slower in one case). Compared with ObliviStore, which handled the applications much more comfortably, the most prominent difference between the approaches is their support for asynchronity, which enables ObliviStore to process requests concurrently. This is critical for performance when the network bandwidth is not a bottleneck. In our research, we found that the network usages of both schemes, under different applications, were well below the bandwidth limit: e.g., with varmail over 4KB blocks, PathORAM consumed about 677 KB/s downstream (and the same upstream), whereas at least 10 MB/s was actually available; ObliviStore consumed about 537 KB/s downstream and 1163.52 KB/s upstream. Although the combined bandwidth was greater than PathORAM, it did not have any observable impact on its performance. In this setting, latency is the actual bottleneck due to the relatively small amount of data that needs to be transferred per request.

Intuitively, it is imaginable that asynchronicity enables an ORAM scheme to work on multiple requests concurrently,

as also identified in the prior research [20]. Our research, however, further reveals the *unique* importance of this property to oblivious cloud storage. The root cause here is that multiple ORAM accesses are required to retrieve a single application object, when the latter is scattered in multiple blocks. This inevitably leads to high response time if the process is not asynchronous (and thus cannot serve multiple ORAM accesses concurrently). Specifically, we replayed the varmail trace without ORAM (i.e., directly on S3) while enforcing synchronous replay, i.e., the next request was not started until the previous one completed. Surprisingly, this did not result in an observable slowdown. However, when we split each object into blocks and ran the process again (still without using ORAM), we observed a significant slowdown (e.g., 2.18 for 4KB blocks) in most cases. Observing similar phenomena with other applications, we learn that asynchronicity is critical for services that access multiple blocks to retrieve an object, which is common in applications of cloud storage systems.

Given the importance of asynchronicity, one may ask how easily it can be achieved. This turns out to be more complicated than it appears to be, and there are many pitfalls when it comes to implementation. Take ObliviStore [20], the most promising prior work that supported asynchronicity, as an example. Although the paper formally proved the security of its construction in handling asynchronous operations, we discovered a subtle issue that apparently undermined its security claims. The problem stems from concurrent processing of interdependent reads/writes, such as multiple requests of the same ORAM block. The implementation of ObliviStore used an "AsyncConflictScheduler" to sequentialize the processing of requests targeting the same ORAM block. However, an adversarial cloud server could still observe the difference between the processing times of a sequence of highly interdependent requests and that of independent requests. To understand whether this would indeed leak information, we conducted an experiment using our implementation of ObliviStore in Java, which performs an equivalent sequentialization of such conflicting requests. We used two sequences of 1000 requests (with identical requesting timestamps): the first randomly read blocks 1 through 1024, whereas the second read blocks 1 through 4. We then repetitively replayed the two sequences 15 times. It turned out that a two-sample Kolmogorov-Smirnov test (which returned a p-value in the order of 10^{-7}) easily distinguished the two sequences based on their different processing times: 18.12 seconds on average for the first versus 36.39 seconds for the second. This issue was not discussed in the ObliviStore paper [20] and appeared to be a realistic attack of the original C# implementation of ObliviStore.

Bandwidth asymmetry. As mentioned earlier, we observed significant differences in the per request response time across different ORAM schemes (Table 1), even when the bandwidth overheads are comparable (see Figure 2a). For example, for each request, PathORAM always downloads all the nodes along a path and then uploads the nodes on the same path, whereas ObliviStore downloads one block per non-empty layer, and uploads those blocks back later, during its background shuffling operation. Due to the presence of dummy blocks, ObliviStore tends to upload more data than it downloads on average, per request (e.g., 176.4 KB downloaded vs 392.7 KB uploaded, for varmail with 16KB blocks). This has an observable impact on the monetary cost to support cloud applications. In particular, for the same ORAM capacity the monetary cost of operating PathORAM is only slightly higher than that of ObliviStore, even though the latter uses less than a fourth of the bandwidth. This is because cloud storage systems tend to charge much more for uploading data than downloading. For S3, the cost of a `put` operation is 12.5 times that of a `get` operation.

		1KB	2KB	4KB	8KB	16KB	32KB	64KB	128KB
GET	Median	59.0	59.0	60.0	62.0	61.0	62.0	62.0	91.0
	IQR	11.5	10.0	10.5	13.0	11.5	20.5	14.0	28.5
PUT	Median	87.5	91.0	87.0	96.0	102.0	107.0	111.5	136.0
	IQR	17.5	16.0	17.0	18.0	25.0	20.0	35.0	49.0

Table 4: Median and interquartile range of the response time (ms) for varying object sizes of single object PUT, GET operations on S3.

Beside financial cost, we may ask whether there is any performance difference, in terms of response time, between the `get` and `put` operations. For this purpose, we ran a set of experiments directly on S3 (without ORAM) that performed the operations on blocks of varying sizes (from 1KB to 128KB). For each size, we created a bucket and populated it with 1024 blocks of that size, each containing random data. For each operation and each block size, we performed the operation on a randomly chosen block (among the pool of 1024 objects in the bucket). The experiment was repeated 200 times, and the response time for each request was measured. Table 4 shows the median and interquartile range of the response time (in ms). The response time is dominated by the network latency, especially when the block sizes are small (e.g., $\leq$ 16KB). The median response time for a `put` operation was between 1.45 and 1.8 times that of its `get` counterpart. This demonstrates that `put` operations are not only more expensive financially but also significantly slower. The finding shows that compared with the overall bandwidth consumption, download and upload bandwidths need to be considered separately for a more realistic measurement of an ORAM scheme's performance.

Data access strategies. The pattern of API calls to the cloud storage server may also affect the ORAM response time. For instance, PracticalOS downloads a single block per request, but must perform a full reshuffling of all the storage blocks every $\sqrt{n}$ requests, hence resulting in longer average response time due to the high cost of the reshuffling. Another less extreme case is PathORAM, whose unit of data access are nodes (i.e., a set of blocks). It issues $\log n+1$ requests to download (or upload) a path, instead of $z(\log n+1)$ requests (where z denotes the number of blocks in a node). Thus one may ask: is it the case that accessing a fixed amount of data (say 64KB) results in the same delay regardless of whether it is accessed in smaller or bigger chunks[7]?

To answer this question, we first ran experiments that performed `get` and `put` operations on 4KB blocks, where the number of *simultaneously issued* operations varied from 1 to 16. Here the response time measured the interval between the initiation of the first operation and when all operations are completed. Table 5 shows the median and interquartile range of the response time (in ms) for different numbers of simultaneous requests: as the number of simultaneously

[7] In both cases, all operations can be performed in parallel using separate threads to minimize the overall response time.

Number of objects		1	2	4	8	16
GET	Median	60.0	64.0	74.0	83.5	115.5
	IQR	10.5	15.0	40.0	37.0	78.5
PUT	Median	87.0	93.0	106.0	137.5	164.0
	IQR	17.0	34.5	57.5	81.0	81.5

Table 5: Median and interquartile range of the response time (ms) for varying number of PUT, GET operations on S3, with 4KB objects.

queried blocks increases, so do the response time and its interquartile range. We note that the increase was not due to scheduling: the median scheduling time was under 10ms, even for simultaneous query of 16 blocks.

Time (ms)	Op	1×64KB	2×32KB	4×16KB	8×8KB	16×4KB
Median	GET	63.0	67.0	85.0	91.0	128.5
	PUT	125.0	132.0	129.5	153.5	176.5
75th percentile	GET	77.0	84.0	117.5	131.5	184.5
	PUT	199.5	172.5	184.0	209.5	234.5
90th percentile	GET	110.0	127.0	187.5	202.0	270.0
	PUT	239.5	229.5	239.5	301.0	319.5

Table 6: Response time (ms) for simultaneous GET, PUT operations of varying number of objects of varying size, totaling 64KB in each case, on Amazon S3.

We then ran another set of experiments to assess the response time of accessing 64KB data in varying sizes of blocks (which are requested simultaneously). Table 6 shows the median, 75th and 90th percentiles of the response time. We found that it is significantly faster to access fewer-but-larger blocks than multiple smaller one, given that their total sizes remain the same. For example, the median response time for downloading 16 blocks of 4KB each was 51% and 92% longer than getting 4 blocks of 16KB each, and 2 blocks of 32KB each, respectively.

These findings suggests that simulation-based evaluation of ORAM would not reflect the performance of the protocol over real-world cloud storages. Unfortunately, it is common in the literature, for instance, to assume a storage backend whose response time is 50ms, independent of the block size, type of operation (`get` or `put`) and data access strategy.

5. METRICS AND THE GAP

Next, we summarize a list of important metrics in evaluating oblivious storage schemes and highlight the gap exhibited by these metrics.

5.1 Metrics for Oblivious Cloud Storage

Response time and slowdown. The response time captures the per-request delay experienced by the application. The slowdown measures the ratio of the time taken by an application to process a batch of operations on top of ORAM and that of processing the same batch without ORAM. Both metrics need to be taken into account: the response time alone is insufficient as it does not capture the impact of asynchronicity. Though synchronous schemes like PathORAM may exhibit low response times, their slowdowns can still be high since all requests are processed sequentially.

Monetary expense. The operating monetary cost, though overlooked by the literature, is a key metric for ORAM. Indeed, it is one of the most important reasons that people opt to use cloud services. The monetary cost is not always strongly correlated with the network bandwidth usage, as designs with low bandwidth overhead may actually incur higher monetary expense. On the other hand, monetary expenses subsumes some traditional metrics, such as server-storage overhead.

Local storage and outsource ratio. Various ORAM designs minimize the amount of local storage. This is out of touch with the goal of oblivious cloud storage, and we argue a better goal is to aim for reasonable outsource ratio (e.g., 1TB stored on the cloud for 1GB stored locally). Indeed, workstations and laptops have hundreds of GB of available local storage, even smartphones have dozens.

To understand the cost of minimizing local storage, we ran an experiment using PathORAM. which involves a linear-size position map that can be stored recursively on the cloud (as opposed to being kept locally). We ran two experiments consisting of replaying a varmail trace of 200 requests (with 4KB blocks and 1GB ORAM capacity, i.e., $n = 2^{18}$ blocks). In the first case, the position map (about 756KB) is stored locally, whereas in the second it is stored recursively on the cloud. Without recursion, the outsource ratio was 929, response time 1.712 sec, and slowdown 9.5; whereas with recursion, the outsource ratio was 2869, response time 4.344 sec, and slowdown 24.1. That is, the cost of storing the position map recursively is: 2.54 times higher response time and slowdown, a 1.86 times higher monetary cost, all for a 3 times higher outsource ratio.[8]

Elasticity. An essential feature of the cloud is its elasticity. While this may not be directly critical for performance, real-world applications may expect such a feature.

Reliability. Cloud applications using S3 as a storage backend require that files are stored durably and reliably. This is especially true for backup or file storage apps. However, many ORAM schemes keep local state that is not replicated on the storage, e.g., position maps, cryptographic keys, and blocks in a local cache. As a result, the reliability guarantee provided by the cloud is forfeited because critical local data may be lost due to client-side software or hardware failures.

5.2 Are We on the Right Track?

In this section, we analyze the ORAM literature to understand if we are moving in the right direction.

Methodology and paper collection. We collected papers which cited the four ORAM designs, and among those, papers which proposed (or improve upon) an ORAM scheme.

Analysis and discussion. At the time of writing, the citation numbers of the four schemes considered in this paper are: 91 (PathORAM, 2013), 47 (ObliviStore, 2013), 61 (LayeredORAM, 2011), and 28 (PracticalOS, 2012). We found a total of 39 papers citing the four schemes we considered which proposed new ORAM schemes or improvements. These 39 papers cited PathORAM 30 times, ObliviStore 16 times, LayeredORAM 23 times, and PracticalOS 10 times. While tree-based ORAMs have received the most attention (and PathORAM is the most cited scheme considered), we

[8] To achieve a constant outsource ratio, storing the position map locally makes sense as long as the ratio between the block size (in bits) and $\log_2 n$, is greater than or equal to the desired outsource ratio.

found it less suitable for cloud storage than ObliviStore (see Section 4).

Also, out of the 39 papers, we found 29 papers mentioning asynchronicity and concurrent requests processing, 14 with some discussion of those properties, but only 3 actually supported such capability with an implemented system or simulation (all 3 are partition-based and by the same research group). Of these, 2 were published in 2013 (out of 9 ORAM papers) and only 1 was published in 2014 (out of 14 ORAM papers). It seems asynchronicity has not become the research focus of the community; though, we acknowledge that many ORAM papers do not consider cloud storage as their primary setting. Finally, we found only one paper [3] which considers the impact of the application/workload in the cloud storage setting, albeit in simulations.

Our findings suggest that ORAM has improved so much that practical performance considerations ignored before (in favor of asymptotic improvements) should now be taken into account. We also stress that our results may not be applicable to scenarios outside of cloud storage. Nevertheless, we believe that our methodology to understand the gap between theory and practice may also be of value for other scenarios.

6. THE CURIOUS FRAMEWORK

In this section, we propose CURIOUS a general partition-based framework for oblivious cloud storage, which we argue and show experimentally is thus far the most promising approach.

6.1 Design Goals

Sections 4 and 5 indicate that: (1) metrics such as bandwidth overhead or client storage, treated as paramount in the existing ORAM literature often fail to capture real world performance; (2) other important metrics and properties (e.g., monetary expense, reliability) have been overlooked; (3) even when targeting the right set of metrics and properties, implementation usually involves several subtle issues (that can result in security vulnerabilities such as the one described in Section 4.2) and require major software engineering effort.

Following our findings, we incorporated the metrics described in Section 5.1 into the design goals of CURIOUS. Additionally, we emphasize the following:

1. Compatibility with storage APIs: schemes must use the APIs of existing cloud storage systems, which *do not offer any ability of computation*, since existing cloud storage services are unlikely to change their APIs to suit ORAM.
2. Asynchronicity: schemes must be able to process multiple requests concurrently, so as to offset the adverse impact of network latency.
3. Elasticity: schemes should be able to grow and shrink the capacity of the storage in inexpensive ways.
4. Reliability: schemes should be able to efficiently recover from client crashes (e.g., corrupted client caches) without data loss.
5. Parameterized local storage: schemes should be able to tune the amount of local storage used, since, for typical application scenarios, only the actual outsource ratio matters.

6.2 Design and Implementation

We describe the design of CURIOUS, a modular partition-based framework, which despite being asymptotically worse (in terms of bandwidth overhead) is able to outperform ObliviStore in both monetary expense and response time, by exploiting some observations of the previous sections.

Modular partition-based framework. CURIOUS utilizes many small constant-size ORAMs (called subORAMs, or partition ORAMs), and uses existing remote storage services in a black-box way. At a high level, the framework consists of a position map, an eviction cache (both stored locally), and a collection of m subORAMs (whose state is kept locally, but whose storage is outsourced to the cloud). CURIOUS is modular: it cleanly separates the modules so that modules (e.g., partitions) can be improved upon independently, and specific modules may be replaced by others in order to suit a specific application scenario.

To process a request for block x, CURIOUS uses the position map to find which subORAM contains x. It then calls the subORAM module to both retrieve x and evict one or more (possibly dummy) blocks to that subORAM. Once retrieved, x is put into the eviction cache, and associated with a random subORAM. This ensures x will be evicted at a random time, preventing the cloud from learning information about the requests from the subORAM access sequences.

Comparing to ObliviStore. CURIOUS resembles ObliviStore in its partition-based design. However, we made substantial improvements to meet the additional design goals.

ObliviStore uses a variant of layered ORAM in each partition and includes multiple optimizations such as: background shuffling, batched shuffling (multiple shuffle jobs can be batched into a single job), and cache-ins during shuffling. In contrast, CURIOUS performs evictions (which may lead to shuffling depending on the subORAM technology) directly after each access. This significantly simplifies the scheme.

ObliviStore makes use of 4 semaphores (only 2 found in the implementation) to synchronize the background shuffling and handling new requests, which involves a complex procedure to avoid timing leaks. CURIOUS bounds the number of requests processed concurrently using a single semaphore (initialized to 128, in our experiments) which is decremented when a request is scheduled, and incremented once the last storage operation associated with the request completes.

Another significant difference lies in the asynchronous processing of concurrent requests. ObliviStore does not define a concurrency model (i.e., under what circumstances two requests can run concurrently) and must ensure that concurrent requests are supported even when a partition is being reshuffled. This may lead to the security issue detailed in Section 4.2. It is unclear how (or at what cost) this problem can be fixed while still retaining other optimizations of ObliviStore.

Asynchronicity and concurrency. Oblivious processing of concurrent requests is challenging, and can compromise security or correctness when done incorrectly. To illustrate this, consider the sequence of requests `get`, `put`, `get`, all for the same block x. If an asynchronous scheme processes these requests sequentially, whereas other requests would be processed concurrently, it becomes vulnerable to certain attacks (Section 4.2). However, naively processing the three

Local state:
1. Position map, storing for each block k: a pair (s, p), where s is the subORAM, and p the position within that subORAM.
2. Eviction cache, storing pairs (k, s), where k is a block and s is the index of the associated subORAM.
3. For each of the m subORAMs: their state.

`scheduleGet`(k, callback)
- Call `schedule`(k, $\perp$, callback).

`schedulePut`(k, v, callback)
- Call `schedule`(k, v, callback).

`lookupPos`(k)
- Lookup (s, p) for block k.
- If block k is in cache, set s and p uniformly at random.

`evictBlocks`(s)
- Pick and remove c cache entries of the form (k, s).
- If there are less than c such entries, add dummy blocks.

`addToCache`(k)
- Pick s, the index of a subORAM, uniformly at random.
- Store (k, s) in the cache.

`schedule`(k, v callback)
- Call `lookupPos`(k) to get (s, p).
- Call `evictBlocks`(s) to get e, the set of blocks to evict.
- Call s.`retrieveBlockAndEvict`(p, e).
- If $v \neq \perp$, then overwrite then content of block k with v.
- If block k is newly retrieved, call `addToCache`(k).
- Call callback.

Figure 3: Modular framework construction.

requests concurrently can compromise correctness or security, too. Indeed, naive processing would, for each request, lookup the position of x (i.e., which subORAM) using the position map, and then it would retrieve x. Now the cloud observes three concurrent requests to the same subORAM, an event that would happen only with probability $1/m^3$ for m subORAMs (e.g., $m = 2^{10}$), if the three requests were independent. In addition, we need to ensure correctness, i.e., the last request (`get`) must return the data written to the block by the 2nd request (`put`), and consistency, e.g., a request should not unexpectedly override something written by a concurrent request.

To address these requirements, CURIOUS leverages modularity and adopts a simple concurrency model: the event that any two requests are run concurrently is statistically independent of their requests' parameters (i.e., type and block). To ensure this, CURIOUS uses a sequential scheduling process that detects whether two requests are "in conflict" (e.g., they access the same block). By keeping track of pending requests, the framework can make such conflicts oblivious to the cloud (i.e., it appears as if such conflicting requests are of any two random requests).

Construction. Figure 3 describes the modular construction of CURIOUS. The interface includes `scheduleGet` and `schedulePut`, both of which are asynchronous (i.e., the call returns immediately, but the callback is invoked upon completing the request).

To process requests concurrently as well as obliviously, CURIOUS ensures that the event that any two requests are processed concurrently is statistically independent of those requests. Each request will operate on a random subORAM so two requests are only competing (i.e., must be executed sequentially) if they operate on the same subORAM. When a request operates on a subORAM, it gets a lock on it and only accesses blocks of that subORAM.

We schedule two competing requests sequentially, though the request processing (i.e., accessing the subORAM) can be asynchronous and concurrent. The idea is that when scheduling a request, the framework will mark the targeted block as "in transit", indicating that a request is in the process of retrieving that block. Subsequent requests for the same block are aware of the fact that the block is already being retrieved so they perform a dummy access (to a random subORAM) to hide (to the cloud) the fact that the two requests targeted the same block. Upon finishing the first request, the block is put in the cache, and also delivered to each concurrent request targeting the same block. This prevents the kind of leaks uncovered in Section 4.2, at the cost of disallowing requests to concurrently operate on the same subORAM.

To address correctness, i.e., it must appear (to the application) as if requests are processed sequentially, CURIOUS maintains a version id in the header of each block. This allows the framework to ensure that every `get` always retrieves the correct version of the data, even in presence of concurrent `put`s to the same block.

Eviction. Akin to ObliviStore, when a block is added to the eviction cache (e.g., as a result of a request) it becomes associated with a uniformly randomly chosen subORAM. This random choice is integral to ensure obliviousness. CURIOUS performs evictions right after each subORAM access, i.e., a constant number of blocks associated with that subORAM (in the eviction cache) are re-written to it. Dummy blocks are used for padding, if needed. The subORAM module decides how many blocks are evicted.

To the cloud, the eviction process is statistically independent of the requests, because all it sees is a fixed number of blocks (some of which may be dummy) evicted to the same subORAM that was just accessed. This is guaranteed if the subORAM's module eviction process (in terms of what the cloud sees) is also independent of which blocks are evicted.

Security. Part of the challenge to secure an asynchronous scheme involves timing. There are two ways in which timing may leak information: (1) through the application running on top of ORAM whose requests have input-dependent timing patterns, and (2) due to a weakness in the ORAM design itself. Like ObliviStore, CURIOUS only addresses the latter. The former is not meant to be prevented by ORAM, which was not designed to hide application timing. We adopt the security definition of ObliviStore (Definition 1), which roughly says that for any two applications with the same timing pattern, the ORAM's timing and accesses must be statistically indistinguishable.

To prove the security of CURIOUS we need to show that what is observed by the cloud provider is statistically independent of the requests (i.e., type, blocks, timing). Consider a CURIOUS instance of a fixed capacity with m subORAMs. The cloud sees: (1) the timing of operations to the storage, (2) the sequence of subORAMs accessed, and (3) the exact operations to each subORAM's raw storage.

CURIOUS' concurrency model (as argued above) takes care of (1). Regarding (3), we argue for the statistical independence of it with the requests after introducing the proposed subORAM design. This leaves (2) to be dealt with here.

THEOREM 6.1. *For any sequence of t requests, the sequence of subORAMs accessed by CURIOUS is statistically independent of the type and target block of those requests.*

PROOF. Consider an arbitrary request; there are three possibilities for the targeted block: (1) it has not been re-

Storage organization:

1. A b-ary tree of depth d where each node has a capacity of z blocks.

`retrieveBlockAndEvict`(p, e)

- Retrieve path ending at leaf p from storage.
- For each block $\in e$:
  - Pick a random leaf l.
  - Update the position map with its new leaf l.
- Rewrite path p, pushing blocks down to the leaf as far as possible.

Figure 4: Tree-based SubORAM.

quested before; (2) it has been requested before and is in the cache; or (3) it has been requested before and is not in cache. Since during initialization blocks are randomly assigned to a subORAM, for (1), from the point of view of the cloud, a uniformly random subORAM will be accessed. For (2), the block is in the cache, so a uniformly random subORAM will be accessed. Finally for (3), the block is not in the cache, so it must have been evicted earlier, at which time a uniformly random subORAM was accessed. □

SubORAM designs. A subORAM module defines a single function: `retrieveBlockAndEvict`, which retrieves a block (given its position information) and evicts a list of blocks in one operation. The module is a tree-based ORAM, which resembles PathORAM [22]. Thanks to bandwidth and cost asymmetry, as well as the data access strategy trait (i.e., it is faster and less expensive to access fewer but bigger chunks of data) of cloud storage system, we found PathORAM fits better to the subORAM design than LayeredORAM and PracticalOS. The construction, shown in Figure 4, makes use of a b-ary tree (for any $b \geq 2$) whose nodes are buckets containing z blocks, for a small integer z.

The security of this subORAM design can be easily derived from that of PathORAM. Namely, blocks written to a subORAM are associated with a uniformly random leaf, hence a random path is accessed for each ORAM read/write. Note that, there is no stash associated with the subORAM, instead when a path overflows, we add the overflown blocks to the client cache. We can choose values of b and z, such that the outsource ratio remains satisfactory. Additionally, in order to exploit the download/upload asymmetry, we can deterministically re-write only the first half of the path some of the time, e.g., for the first out of every two re-writes, so as to lower the average number of uploaded nodes per request (at the cost of lowering the outsource ratio, since blocks are more likely to overflow).

6.3 Extensions

Elasticity. CURIOUS supports growing/shrinking its capacity in two ways. The first way supports elasticity independent of the subORAM module. To increase the capacity, the framework adds a new subORAM instance, randomly re-associates blocks in the eviction cache, and proceeds to build the subORAM locally until it can be written to the storage. This is done as in ObliviStore [20] by moving some random blocks from the existing subORAMs to the new instance. Randomly re-associating blocks in the cache with uniformly chosen subORAMs hides which blocks from the cache (if any) will be evicted to the new subORAM. The fact that a new subORAM was allocated is not hidden. In order to shrink the capacity, before a subORAM is deleted, all its blocks must be retrieved from the storage and added to the eviction cache, after which we re-associate the blocks in the cache with uniformly randomly chosen subORAMs.

The second way to achieve elasticity comes from the subORAM module, assuming the subORAM itself supports it. Layered ORAMs naturally support both adding a new layer (to increase capacity) and merging a layer down to lower layers, provided all blocks fit in the lower layers (to reduce capacity). Recent work has shown that tree-based designs [13] can also be augmented to support grow/shrink operations.

Note that elasticity leaks an upper bound on the capacity of the scheme. To avoid leaking more than this, we can let the application ask for an increase/decrease in capacity, and instead of performing such operations right away, we can do them lazily over time, or at random times.

Reliability. ORAM state, i.e., the encryption key, the position map, and the eviction cache, which are stored locally, may be lost when a failure occurs (e.g., hardware failure). The encryption key can be derived from a secret (e.g., password). The position map can be reconstructed by reading the entire storage (with block headers that store a block id and version id). However, some blocks in the eviction cache may not have a copy on the cloud, so that they would be lost if a failure occurs.

We propose to backup this information obliviously, during the processing of requests, by making use of a circular buffer of size $O(m\gamma)$, where m is the number of subORAMs, and γ is a parameter. (We set $\gamma = O(\log m)$, or $\gamma = \omega(\log m)$, depending on the desired bound on the probability of data loss.) The idea is that with each request, we backup, to the circular buffer, a small (constant) number of blocks newly added to the cache. This works because processing each request results in only a small number of new blocks added to the cache (typically 0 or 1). If each request backs up the previous request's new blocks, the risk of data loss is minimized. In the event of a failure, the data blocks can be retrieved by reading the entire circular buffer.

6.4 Evaluation

We evaluate CURIOUS against ObliviStore using the same experimental setup as in Section 4. We replay the application traces, with capacity 256MB and block size 16KB, in all cases. For a fair comparison, we set the number of subORAMs of CURIOUS such that both schemes have roughly the same outsource ratio (or ObliviStore has the advantage). The results, displayed in Table 7, show that CURIOUS is better in supporting the selected applications, i.e., its slowdown is either the same or less than ObliviStore, despite the more than doubled bandwidth usage in some cases (e.g., for fileserver, the bandwidth usage of tree-based CURIOUS is 4500.4 KB/req vs. only 2159.8 KB/req for ObliviStore). Further, we see that the monetary expense incurred by CURIOUS is between 1/2 and 2/3 that of ObliviStore. In terms of applications, both varmail and webproxy are easily supported by both schemes, but CURIOUS has slightly higher response time (e.g., 266 ms vs. 200ms for webproxy). This is due to the background shuffling of ObliviStore which, for less demanding applications, is beneficial because the shuffling cost is not paid upfront. For such applications, minimizing the response time below a certain threshold may not be required; instead in such cases, monetary expenses may outweigh small differences in response times. For these applications, CURIOUS' operating monetary cost is almost half of ObliviStore. Also, the comparison is conservative

		varmail	webproxy	webserver	fileserver
	Metric	16 KB	16 KB	16 KB	16 KB
ObliviStore	Bandwidth usage	623.0	590.8	454.9	2159.8
	Response time	0.196	0.200	7.950	13.531
	Slowdown	1.000	1.000	1.152	3.767
	Monetary cost	0.153	0.144	0.113	0.548
	Outsource Ratio	171.2	99.5	16.1	18.5
CURIOUS tree-based	Bandwidth usage	1025.0	988.7	852.5	4500.4
	Response time	0.270	0.266	2.004	2.114
	Slowdown	1.000	1.000	1.114	2.702
	Monetary cost	0.081	0.078	0.067	0.355
	Outsource Ratio	209.7	111.3	25.6	18.7

Table 7: Replaying application traces with ObliviStore and CURIOUS, all in 16KB blocks, and according to various metrics: bandwidth usage (in KB/req), mean response time (sec/req), relative slowdown, monetary cost (in USD / 1000 req), and outsource ratio. 1000 requests for both varmail and webproxy, 2500 for webserver, and 750 for fileserver. For CURIOUS tree-based, we set $b = 3$ and $z = 5$.

as ObliviStore's performance may degrade after the security issue on handling concurrency (Section 4.2) is fixed.

For the demanding applications (i.e., webserver and fileserver) which stressed ORAMs, CURIOUS is a better fit than ObliviStore. Indeed, due to background shuffling, and high upload cost, ObliviStore experienced high response times and larger slowdown. Take webserver as an example, the response time was almost 4 times that of CURIOUS (i.e., 7.950 sec/req for ObliviStore vs. only 2.004 sec/req for CURIOUS) but the slowdown was comparable; both schemes were close to being able to fully support the application. For fileserver, though neither scheme is even close to satisfying the demands of this application, nevertheless CURIOUS significantly outperformed ObliviStore both in response time and slowdown.

7. CONCLUSIONS

In this work, we systematically evaluated four representative ORAM designs on Amazon S3. We replayed traces that represent typical workloads of applications like web servers, mail and file servers. We found that existing ORAM designs are unable to support such applications, partly due to mismatches between ORAM theory and practical performance and requirement of cloud applications. We converge to a set of metrics and important properties, and describe CURIOUS, a modular framework which is the most promising approach thus far to oblivious cloud storage.

8. ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation under Grant Number CNS-0964392, 1223967, 1117106, 1223477, 1223495, 1408874, and NSF-1464113.

9. REFERENCES

[1] D. Boneh, D. Mazieres, and R. Popa. Remote Oblivious Storage: Making Oblivious RAM Practical. 2011.

[2] K. Chung, Z. Liu, and R. Pass. Statistically-Secure ORAM with $\tilde{O}(\log^2 n)$ Overhead. *ASIACRYPT*, 2014.

[3] J. Dautrich, E. Stefanov, and E. Shi. Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns. In *USENIX Security*, volume 14, 2014.

[4] S. Devadas, M. van Dijk, C. W. Fletcher, and L. Ren. Onion ORAM: A Constant Bandwidth and Constant Client Storage ORAM (without FHE or SWHE). IACR ePrint, 2015.

[5] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 1996.

[6] M. T. Goodrich. Randomized Shellsort: A Simple Oblivious Sorting Algorithm. In *SODA*, 2010.

[7] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM Simulation with Efficient Worst-Case Access Overhead. In *CCSW*, 2011.

[8] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical Oblivious Storage. In *CODASPY*, 2012.

[9] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (In)security of Hash-Based Oblivious RAM and a new Balancing Scheme. In *SODA*, 2012.

[10] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *CCS*, 2013.

[11] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient Private File Retrieval by Combining ORAM and PIR. In *NDSS*, 2014.

[12] R. McDougall. Filebench Tutorial. 2004.

[13] T. Moataz, T. Mayberry, and E. Blass. Resizable Tree-Based Oblivious RAM. *IACR ePrint*, 2014.

[14] T. Moataz, T. Mayberry, and E.-O. Blass. Constant Communication ORAM with Small Blocksize. In *CCS*, 2015.

[15] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal. The Melbourne Shuffle: Improving Oblivious Storage in the Cloud. In *ALP*. 2014.

[16] B. Pinkas and T. Reinman. Oblivious RAM Revisited. In *CRYPTO*, 2010.

[17] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security*, 2015.

[18] S. Shepler, E. Kustarz, and A. Wilson. Filebench. `http://sourceforge.net/projects/filebench/`, 2015.

[19] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $o((\log n)^3)$ Worst-Case Cost. In *ASIACRYPT*, 2011.

[20] E. Stefanov and E. Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *S&P*, 2013.

[21] E. Stefanov, E. Shi, and D. X. Song. Towards Practical Oblivious RAM. In *NDSS*, 2012.

[22] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *CCS*, 2013.

[23] X. S. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. IACR ePrint, 2014.

[24] P. Williams and R. Sion. Single Round Access Privacy on Outsourced Storage. In *CCS*, 2012.

[25] P. Williams, R. Sion, and A. Tomescu. PrivateFS: A Parallel Oblivious File System. In *CCS*, 2012.