Multi-Cloud Oblivious Storage

Emil Stefanov
UC Berkeley
emil@cs.berkeley.edu

Elaine Shi
University of Maryland
elaine@cs.umd.edu

ABSTRACT

We present a 2-cloud oblivious storage (ORAM) system that achieves 2.6X bandwidth cost between the client and the cloud. Splitting an ORAM across 2 or more non-colluding clouds allows us to reduce the client-cloud bandwidth cost by at least one order of magnitude, shifting the higher-bandwidth communication to in-between the clouds where bandwidth provisioning is abundant. Our approach makes ORAM practical for bandwidth-constrained clients such as home or mobile Internet connections. We provide a full-fledged implementation of our 2-cloud ORAM system, and report results from a real-world deployment over Amazon EC2 and Microsoft Azure.

Categories and Subject Descriptors

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

General Terms

Algorithms; Security

Keywords

Oblivious RAM; outsourced storage; multi-cloud; privacy

1. INTRODUCTION

Storage outsourcing is a growing industry that shields storage users from the burden of in-house infrastructure maintenance, and offers economies of scale. However, due to concerns over data privacy, "many potential cloud users have yet to join the cloud, and many are for the most part putting only their less sensitive data in the cloud" [8]. Encryption alone is insufficient for ensuring data privacy, since access patterns can also leak a considerable amount of sensitive information. For example, Islam *et al.* demonstrate statistical attacks that leverage access patterns to successfully infer about 80% of the search queries made to an encrypted email repository [18].

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516673.

Oblivious RAM (ORAM) [7, 9–11, 13, 14, 16, 19, 22, 23, 25, 30–32, 34, 35], first proposed by Goldreich and Ostrovsky [11], is a cryptographic construct that allows a client to access encrypted data residing on an untrusted storage provider, while completely hiding data access patterns. ORAM guarantees that the sequence of physical addresses accessed is independent of the actual data requested – in other words, the *physical* access patterns do not reveal information about the *logical* access patterns. To achieve this, the client continuously re-encrypts and reshuffles data blocks on the storage server, to cryptographically conceal the *logical* access pattern.

Due to the attractive security guarantees ORAM provides, several recent works have investigated how to make ORAM practical in an outsourced storage setting [20, 30, 31, 34, 35]. These works suggest that a primary hurdle for real-world ORAM deployment stems from its high bandwidth cost. Under realistic parameterizations (e.g., gigabytes to terabytes of outsourced storage), the best known ORAM schemes incur about 20X – 35X bandwidth cost, i.e., for every data block accessed, on average 20 to 35 blocks need to be transferred. This makes ORAM unsuitable especially for bandwidth constrained clients, e.g., home Internet connections or mobile devices.

In some cases, shipping computation to the cloud can alleviate the client-cloud bandwidth consumption. When the client does not trust the cloud provider, however, one immediate challenge is how to securely compute in the cloud without leaking data to the cloud provider. To this end, researchers have suggested combining trusted hardware and ORAM to securely delegate both computation and storage to the cloud [2, 3, 6, 17, 20, 26]. While this approach is promising, trusted hardware is not widely deployed on today's cloud platforms. Instead, we seek a solution that is *readily deployable* on today's cloud service providers.

1.1 Two-Cloud ORAM

We observe the following asymmetry in bandwidth resource provisioning: while client-cloud bandwidth is scarce in many settings such as cellular networks or slower home Internet connections; network bandwidth in between major cloud providers (e.g., Amazon Web Services and Microsoft Azure) is ample.

We show that by spreading data across multiple *non-colluding* cloud providers, we can shift the 20X – 35X bandwidth cost to inter-cloud communication, while keeping the client-cloud bandwidth cost minimal — about **2.6X** under typical parametrization, and out of which 2X is necessary to hide whether each access is a read or write. While a general k-cloud scheme exists as described in the full online version [29], we focus on designing a highly optimized two-cloud solution which we believe to be the more likely deployment scenario. We provide a full-fledged two-cloud ORAM implementation, and have successfully deployed it on two major cloud providers, Amazon Web Services (AWS) and Microsoft Azure.

While the idea of using multiple non-colluding clouds is natural, how to design a provably secure and practically efficient scheme is challenging. The main challenge is how to *avoid passing data through the client during data shuffling, and still ensuring security against a potentially malicious cloud that can arbitrarily deviate from the prescribed behavior.* Multi-server ORAM has been considered in a theoretic setting by Lu and Ostrovsky [21]. Their construction, however, focuses on asymptotic performance, and still requires passing data through the client during shuffling. Consequently, their construction would result in at least 20X-35X client-cloud bandwidth cost under typical parametrization.

Technical highlights. Our main technique is to let the two clouds shuffle the data amongst themselves. Particularly, one cloud shuffles the data blocks, adds an "onion" layer of encryption, and send them to the other cloud. In this way, one cloud observes the permutation while the other observes the accesses to these blocks — and neither cloud observes both. In the next round, the two clouds switch roles.

To enforce honest behavior when one of the clouds may be malicious, we design a novel *commutative checksum-encryption construction.* At a very high level, the client shares a secret checksum function with each cloud, such that the clouds can monitor each other's actions. After every shuffle, about 4λ checksum bits per block are transferred between the client and the clouds such that the client can verify the correctness of the shuffle. In the above, $\lambda = 128$ is the security parameter. The typical block size is much larger than λ, e.g., 4KB (size of a memory page) or higher. While transferring entire blocks to the client during shuffling would have incurred 20X-35X cost, by transferring 4λ bits per block we bring this part of the cost down to 0.1X to 0.2X (out of a total of 2.6X) as shown in Section 5. We also show that a similar technique may be employed to avoid reading a logarithmic number of blocks for each data access, which is necessary in most existing ORAM schemes.

Similar to the SSS construction [31], our 2-cloud ORAM scheme requires the client to store less than 1.5GB of data for an ORAM of 1TB in capacity (i.e., less than 0.15% of the ORAM capacity). Particularly, in addition to caching about $O(\sqrt{N})$ data blocks where N is the total number of blocks, the client also stores about 4 bytes of metadata per block. While the metadata size is linear in theory; its size in practice is typically comparable to or smaller than the $O(\sqrt{N})$ data blocks cached. Out of theoretic interest, with suitable modifications to our scheme, it is possible to achieve sublinear client storage by applying the recursion technique described in earlier works [27,31], such that we could (recursively) outsource the linear metadata to the cloud as well. In practice, however, this is rarely necessary – and even if one does apply the recursion, the recursion depth rarely exceeds 2 to 3 in practical settings [20, 27].

In addition to minimizing the client-cloud bandwidth consumption, our 2-cloud ORAM scheme also roughly halves the total bandwidth consumption (including inter-cloud and client-cloud communication) in comparison with single-cloud counterparts [31].

Threat model. We assume that at least 1 cloud provider is honest, i.e., faithfully executes the prescribed protocol. The other cloud may be malicious and arbitrarily deviate from the protocol. We do not know ahead of time which cloud is malicious. It is not within our goal to prevent the malicious cloud from launching Denial-of-Service (DoS) style attacks — DoS defense is orthogonal to and outside the scope of this work. We also cannot prevent a malicious cloud from voluntarily divulging its local states and views to the honest cloud, which can potentially allow the honest cloud to learn additional information. For example, a malicious cloud can simply publish its states and protocol views on a public website.

However, we do guarantee that as long as there exists at least one honest cloud, *a malicious cloud cannot learn any information about the client's logical access patterns. In fact, any deviation from the prescribed protocol is immediately detectable by the client.* In other words, by deviating from the prescribed protocol, a malicious cloud can only help an honest cloud gain information, but cannot learn anything itself. Moreover, it will be caught immediately if it ever deviates.

The above intuitive security model is generalized to the k-cloud setting (when only 1 out of k clouds needs to be honest), and defined formally using a simulation-based definition in the full online version [29]. We prove that our constructions are secure under this simulation-based definition in the full online version [29].

2. PRELIMINARIES

Our algorithm depends on the partitioning framework and the partition ORAM construction proposed by Stefanov, Shi, and Song [31], referred to as the SSS construction. We give a brief background of the SSS construction.

It is important to note that partitions do not represent clouds. In fact, as will be later explained in our construction, each partition is split across multiple clouds.

2.1 Partitioning Framework

The SSS partitioning framework [31] allows us to securely break-up ORAM read/write operations into read/write operations over smaller partitions, where each partition is an ORAM itself.

The framework consists of two main techniques, *partitioning* and *eviction.* Through partitioning, a bigger ORAM instance of capacity N is divided into $O(\sqrt{N})$ smaller ORAM instances (called partitions), each with capacity $O(\sqrt{N})$. While naive partitioning can break security, Stefanov *et al.* [31] propose a novel approach to allow partitioning without compromising security, as outlined below.

Figure 1 illustrates the partitioning framework. At any point of time, a block resides in a random partition. The client stores a *position map* to keep track of which partition each block resides in. To access a block whose identifier is u, the client first looks up the position map and determine block u's current partition p. Then the client issues an ORAM read operation to partition p and looks up block u. On fetching the block from the server, the client logically assigns it to a freshly chosen random partition – without writing the block to the server immediately. Instead, this block is temporarily cached in the client's local *eviction cache.*

```
The partitioning framework [31]
// Divide the ORAM into $\sqrt{N}$ partitions of size $O(\sqrt{N})$.

Read(u):
• Look up position map and determine that u is assigned to partition $p$.
• If u is not found in eviction caches:
      – ReadPartition($p$, u)
  Else if u is found in local eviction caches:
      – ReadPartition($p, \perp$)   //read dummy
• Pick a random partition $p'$, add the block identified by u to the eviction caches, and logically assign u to partition $p'$.
• Call Evict $\nu$ times where $\nu > 1$ is the eviction rate.

Write(u, $B$):
Same as Read(u), except that the block written to the eviction cache is replaced with the new block.

Evict:
• Pick a random partition $p$.
• If a block $B$ exists in the eviction cache assigned to partition $p$, call WritePartition($p, B$).
• Else, call WritePartition($p, \perp$), where $\perp$ represents a dummy block.
```

Figure 1: The SSS partitioning framework [31]. Our construction uses this framework to express ORAM **Read** and **Write** operations in terms of the **ReadPartition** and **WritePartition** operations of our multi-clouds construction.

A background eviction process evicts blocks from the eviction cache back to the server in an oblivious manner. With every data access, randomly select 2 partitions for eviction. If a block exists in the eviction cache assigned to the chosen partition, evict a real block; otherwise, evict a dummy block to prevent leakage.

2.2 Partition ORAM

Each partition is a fully functional ORAM in itself. Our partition ORAM construction is based on the partition ORAM of the SSS construction [31], which is a variant of the original hierarchical construction [11], but with various optimizations geared towards maximal practical performance.

Each partition consists of $L := \frac{1}{2}\log N + 1$ levels, where level $i \in \{0, 1, \ldots, L-1\}$ can store at more 2^i real blocks, and 2^i or more dummy blocks. We refer to the largest level, i.e., level $L-1$, as the *top level*.

We extend the client's position map to store the position tuple $(p, \ell, \mathit{offset})$ for each block, where p is the partition, ℓ is the level, and *offset* denotes the offset within the level.

Read. To read a block, the client first reads its position map and find out the position $(p^*, \ell^*, \mathit{offset}^*)$ of the block. Then, the client reads one block from each level of partition p. For level $\ell = \ell^*$, the client reads the block at offset^*. For other levels, the client reads a random unread dummy block. If the block is found in the client's eviction cache, one dummy block is read from each level.

Write. Writing back a block to a partition causes a reshuffling operation. Specifically, let $0, 1, \ldots, \ell$ denote consecutively filled levels such that level $\ell+1$ is empty. Writing back a block B causes levels $0, 1, \ldots, \ell$ to be reshuffled into level $\ell+1$. In the single-cloud SSS construction, the reshuffling is done by having the client download all blocks, permute them, re-encrypt them, and then write them back to the server.

3. BASIC TWO-CLOUD CONSTRUCTION IN THE SEMI-HONEST MODEL

We first explain a scheme that is secure when both clouds are semi-honest. Then, in Section 4, we explain how to use our commutative checksum-encryption construction to achieve security against one malicious cloud (without knowing which one might be malicious).

3.1 Intuition

Reducing ORAM shuffling overhead. In existing single-cloud ORAM schemes [30–32, 35], a major source of the client bandwidth overhead stems from shuffling. Periodically, the client has to download a subset of data blocks from the cloud, permute them, re-encrypt them locally, and write them back.

In our multi-cloud ORAM construction, we delegate the shuffling work to the two clouds to minimize client-cloud bandwidth usage. For example, cloud S_1 shuffles a subset of blocks, adds a layer of encryption (i.e., an "onion layer"), and sends them to S_2. Later when the client needs to read from that subset of blocks, it fetches them from S_2. In this way, while cloud S_1 knows the permutation used in the shuffling, it does not see which blocks the client requests from S_2 later. In contrast, while S_2 sees which shuffled blocks the client requests later, it has no knowledge of the permutation applied by S_1. As a result, neither cloud learns any information about the client's logical (i.e., "unshuffled") access pattern. The details of inter-cloud shuffling are described in Section 3.5.2.

Note that it is necessary for security to add onion encryption layers after every shuffle: if a block B gets shuffled from S_1 to S_2 and then back to S_1, we do not want S_1 to be able to link the two occurrences of the same block B based on the block's data. In the full online version [29], we introduce a background onion removal process to avoid adding an unbounded number of onion layers to a block.

Reducing ORAM read overhead. Existing single-cloud ORAM schemes can also incur significant bandwidth overhead for reading a data block in-between shufflings [30–32, 35]. For example, the client may need to read one or more blocks from $O(\log N)$ levels in a hierarchy.

In our multi-cloud construction, we are able to significantly reduce the bandwidth overhead for reading a block by doing the following. The client requests a set of (already shuffled) blocks from S_1. S_1 then randomly permutes them with a permutation known only by S_1 and the client, and then S_1 sends them to S_2. Finally, the client requests and downloads only a single block from S_2 by providing S_2 with the permuted index of the requested block. The details of reading a block from multiple clouds are described in Section 3.5.1.

3.2 Data Layout on Clouds

Our scheme leverages the partitioning framework as described in Section 2.1. We divide our ORAM into $O(\sqrt{N})$ partitions each of capacity $O(\sqrt{N})$. Each partition is divided into $L = O(\log N)$ levels, and each level resides on one of the two clouds. Which level resides on which cloud changes as partitions are shuffled. Each level can be filled (i.e., it contains blocks) or empty.

In practice, each cloud can distribute the data across multiple servers. For simplicity, we will first regard each cloud as a single logical entity; then in the full online version [29], we explain how to distribute the data and workload across multiple servers within each cloud.

New use of partitioning framework. We use partitioning for a somewhat different motivation than described in the SSS paper [31]. Particularly, the SSS construction relies on partitioning mainly to get rid of oblivious sorting [11,12], thus achieving a constant factor saving in bandwidth.

Like in ObliviStore [30], we use partitioning to achieve parallelism, and to greatly improve response time. We shuffle and read from multiple partitions in parallel as described in the full online version [29], and such parallelism is especially useful if each cloud has multiple servers each serving a subset of the partitions. If a read request happens on a partition not currently being shuffled, the read does not block on shuffling. In our system, the contention (probability of a read occurring on a partition involved in a large shuffle) is very small — since (1) each read happens to a uniformly random partition (regardless of the access pattern), (2) there are thousands of partitions, and (3) large shuffles happen very infrequently, i.e., with exponentially decreasing frequency. It's also important to note that as the amount of parallelism increases, the contention increases. However, in practice, for large enough ORAMs with many partitions, the amount of parallelism necessary to saturate the server bandwidth does not cause significant contention.

In our multi-cloud ORAM, we can still achieve about 2X-3X client-cloud bandwidth cost even with a single partition. However, with a single partition, reads would have to block waiting for reshuffling operations of up to N blocks to finish (e.g., gigabytes or terabytes of data transfered between the two clouds), resulting in prolonged response times. Williams and Sion [35] describe an algorithm for simultaneously reading and shuffling a single-partition ORAM, but we observe that this technique about doubles the cloud-to-cloud bandwidth used because the single partition has twice as many levels than our $O(\sqrt{N})$ sized partitions.

As studied in [31], a larger number of partitions requires a larger amount of client-side cache. Therefore, the number of partitions can be used as a knob to tune the tradeoff between the client-side storage and the response time.

3.3 Client Metadata

The client stores the following metadata. Note that even though the client storage is asymptotically linear, it has a very small constant and is quite small in practice. For example, all of the client storage combined is less than 1.5 GB for a 1 TB ORAM with 4KB blocks (i.e., less than 0.15% of the ORAM capacity). The amount of client storage used is similar to several existing single-cloud ORAM systems [30,31,35].

- The position tuple (p, ℓ, offset) denoting the block's current partition, level, and offset within the level.

- A bit vector for every partition and every filled level, indicating which blocks in the level have been read and (logically) removed.

- A time value for every partition, i.e., the total number of operations that have occurred so far for each partition.

- A next dummy block counter for each partition and each level. The counter is set to 0 when a level is being rebuilt; and incremented when a next dummy element is read. This counter is sufficient to implement the GetNextDummy function mentioned in Figure 4.

It is possible to even further reduce the client storage to $O(\sqrt{N})$ by recursing on the position map as was proposed in [31].

3.4 Initializing

Each partition is initially placed in a random cloud (S_1 or S_2). Each block is assigned to a random partition, and a random offset in the top level (the largest level) of the partition. The client's position map is initialized accordingly.

We assume initially, all blocks have the value zero. This can be done by initializing all blocks on the server to be encryptions of the zero block with appropriate metadata attached. Initialization can be optimized to consume much less bandwidth using a similar technique described in [31].

3.5 ORAM Read and Write Operations

A standard ORAM includes two operations: **Read** and **Write**. As previously mentioned in Section 2 and Section 3.2, our construction uses the SSS partitioning framework [31]. The SSS framework specifies how **Read** and **Write** operations can be securely expressed in terms of *partition ORAM* read/write operations called **ReadPartition** and **WritePartition**. Section 2 explains how this can be achieved. Since we rely on this framework we only need to describe how to implement **ReadPartition** and **WritePartition**.

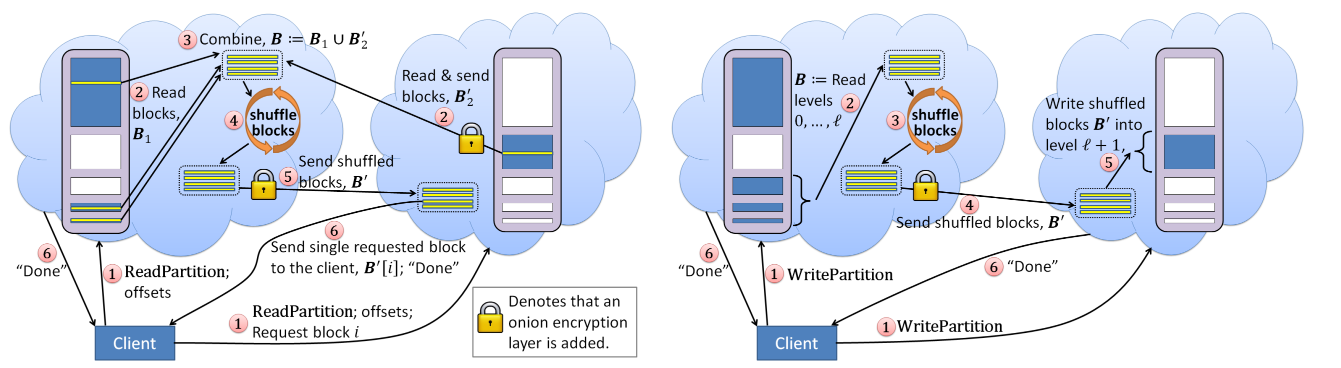
ReadPartition and **WritePartition** operations are performed in parallel across all partitions but in serial for each individual partition to ensure consistency.

3.5.1 Reading from a Partition

In a typical single-cloud ORAM scheme [31], the client needs to download one block for each of the $O(\log N)$ levels in a partition to hide which level it wants to read from. In our construction, we rely on the inter-cloud shuffling technique such that the client only needs to download a single block.

Our **ReadPartition** protocol is illustrated in Figure 2(a) and described in detail in Figure 4. To read a block u, the client first looks up its position map to get the tuple $(p^*, \ell^*, \mathit{offset}^*)$ indicating the position (partition p^*, level ℓ^*, offset offset^*) for block u. The client then generates one offset for each filled level ℓ in partition p. For level $\ell = \ell^*$, the offset $\mathit{offset}_\ell = \mathit{offset}^*$. For all other levels, its offset_ℓ corresponds to a random unread dummy (obtained via GetNextDummy) determined by computing the permuted offset of the $\mathit{nextDummy}[p^*, \ell]$ counter of each level. The client now divides these offsets based on which cloud contains each level, and sends the corresponding offsets to each cloud. *It is important for security that the client sends to each cloud only the offsets of the levels contained within the cloud.*

Now, each cloud reads one block from each of its filled level, at the offsets indicated by the client. The cloud with fewer filled levels onion-encrypts the fetched blocks, and


3 Combine, $B := B_1 \cup B'_2$
2 Read blocks, B_1
4 shuffle blocks
Read & send blocks, B'_2
2
Send shuffled blocks, B'
5
6 Send single requested block to the client, $B'[i]$; "Done"
6 "Done"
1 ReadPartition; offsets
1 ReadPartition; offsets; Request block i
Denotes that an onion encryption layer is added.
Client
$B :=$ Read levels 2 $0, \ldots, \ell$
3 shuffle blocks
Write shuffled blocks B' into level $\ell + 1$, 5
4 Send shuffled blocks, B'
6 "Done"
1 WritePartition
6 "Done"
1 WritePartition
Client

(a) **ReadPartition** (b) **WritePartition** (when $\ell < L - 1$)

Figure 2: Two-cloud ORAM protocol. Actions are performed in nondecreasing order specified by the circled numbers. If two actions have the same number within a diagram, it means that they can happen in parallel.

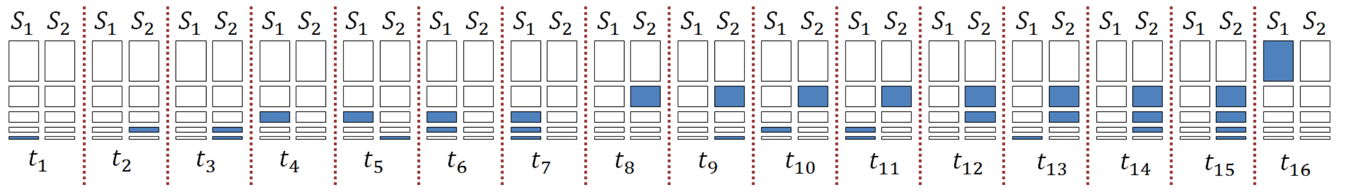


Figure 3: Layout of an ORAM partition across two clouds over time. This figure shows for a specific partition which levels are filled in each cloud (S_1 and S_2) over time ($t_i \in \{t_1, \ldots, t_{16}\}$). When the bottom-most level is empty, our algorithm automatically decides to which cloud to write the next block to preserve the invariant that all consequentially filled levels (from the bottom upwards) are always located on the same cloud.

sends them to the other cloud. Without loss of generality, assume S_2 sends its onion-encrypted blocks to S_1. S_1 now merges the blocks from S_2 with its own fetched set, onion-encrypts them, and reshuffles them using a PRP key shared with the client. The reshuffled set is sent to S_2.

The client now reveals the desired index within the reshuffled array to S_2, and S_2 returns the block at that index. In the full online version [29], we explain how the client decrypts the fetched block. Decrypting involves removing multiple onion layers of encryption as described in the full online version [29].

3.5.2 *Writing to a Partition*

Our **WritePartition** protocol is illustrated in Figure 2(b) and described in detail in Figure 4. Whenever a block is written to a partition p, it is shuffled with all consecutively filled levels $0, 1, 2, \ldots, \ell$ of the partition into level $\ell + 1$. If $\ell = L - 1$ is the top level, all levels will be shuffled into the top level.

Note that there are 2^L possible ways to divide L levels between two clouds. Our assignment algorithm is designed to best facilitate inter-cloud shuffling by enforcing the following invariant.

Invariant. *Any time when consecutive levels* $0..i$ *are filled and need to be shuffled into level* $i + 1$*, levels* $0..i$ *all reside within the same cloud.*

This minimizes the cloud-cloud bandwidth because shuffles always involve consequentially filled levels (i.e., $0..i$). Suppose that, without loss of generality, cloud S_1 holds levels $0..i$ to be shuffled into level $i+1$ of cloud S_2. By ensuring that the above invariant holds, at the start of shuffling, S_1 already has all of levels being shuffled ($0..i$) and does not have to fetch additional levels from S_2 before starting the shuffle. Figure 3 demonstrates how the levels of a partition are divided amongst the two clouds over time.

These shuffled blocks now form level $\ell + 1$ of the same partition, and reside on S_2. Levels $0, \ldots, \ell$ on S_1 of partition p are deleted and future requests for level ℓ of partition p are directed to S_2.

Shuffling the top level. The only exception is when all levels $0..L-1$ are shuffled, i.e., when the top level is involved in the shuffle. In this case, some dummy or obsolete blocks need to be discarded during the shuffle. Therefore, a top-level shuffle needs to be treated slightly differently, where the client tells S_1 (i.e., source of the shuffle) a subset of blocks to keep for each level — without leaking any information. The detailed protocol is described in Figure 4.

Note that almost all of the bandwidth consumed happens when the two clouds exchange blocks, and the client's bandwidth is conserved.

3.6 Onion Encryption Layers

Key generation. The client shares a master secret key msk_1 with cloud S_1; and msk_2 with cloud S_2.

Onion encryption. Whenever a cloud, say S_1, needs to onion encrypt a block, it generates a pseudo-random one-time encryption key based on msk_1, the time value for the partition under consideration, and the block's position tuple pos' after the shuffle[1].

[1]For read-phase shuffles, we use the block's position prior to shuffle.

```
**ReadPartition**($p^*$, u):

                        $C$ :  $(p^*, \ell^*, offset^*) \leftarrow \mathsf{pos}(\mathsf{u})$   // find out where block u resides.
                        $C$ :  Let $offset_{\ell^*} := offset^*$. For $\ell \neq \ell^*$, let $offset_\ell = \mathsf{GetNextDummy}(p^*, \ell)$
                               // The GetNextDummy function returns the position of a random unread dummy in the level
For $j \in \{1, 2\} : C \to S_j$ :  $\{offset_\ell$ : for each filled level $\ell$ in partition $p^* \in S_j\}$
    For $j \in \{1, 2\} : S_j$ :  $\mathbf{B}_i \leftarrow \{\mathsf{BlockAt}\,[p^*, \ell, offset_\ell] : \forall$ filled level $\ell$ in partition $p^*$ on $S_j\}$
                      $S_2$ :  $\forall i \in |\mathbf{B}_2| : \mathbf{B}'_2[i] := E_{ek[i]}(\mathbf{B}_2[i])$ where $ek[i] := \mathsf{PRF}_{msk_2}(i, p^*, t, \text{“read-enc-A”})$
            $S_2 \to S_1$ :  $\mathbf{B}'_2$   // Without loss of generality, assume $S_2$ has fewer filled levels.
                      $S_1$ :  $\mathbf{B} := \mathbf{B}_1 \cup \mathbf{B}'_2$
                      $S_1$ :  $\forall i \in |\mathbf{B}| : \mathbf{B}'[\mathsf{PRP}_{sk}(i)] := E_{ek[i]}(\mathbf{B}[i])$
                               where $sk := \mathsf{PRF}_{msk_1}(p^*, t, \text{“read-shuffle”})$,   $ek[i] := \mathsf{PRF}_{msk_1}(i, p^*, t, \text{“read-enc-B”})$
            $S_1 \to S_2$ :  $\mathbf{B}'$
             $C \to S_2$ :  $i =$ (offset of block u in $\mathbf{B}'$)   // $C$ can compute this with $msk_1$ and $msk_2$
             $S_2 \to C$ :  $\mathbf{B}'[i]$
For $j \in \{1, 2\} : S_j \to C$ :  “Done”
```

```
**WritePartition**($p^*$, $B$):
/* Without loss of generality, assume that cloud 1 needs to shuffle $\ell$ consecutive levels and send to cloud 2. Assume that the block to
be written $B$ has been encrypted by the client using a secret key known only to itself.*/

**Case 1:** $\ell < L - 1$

 $C \to S_1$:   “Write partition”,  $p^*$,  $B$

       $S_1$:   $\mathbf{B} \leftarrow \mathsf{BlocksAt}[p^*, 0..\ell] \cup \{B\}$
       $S_1$:   $\forall i \in |\mathbf{B}|$ :  **do**
                  Let $i' = \mathsf{PRP}_{sk}(i)$,  $\mathbf{B}'[i'] := E_{ek[i]}(\mathbf{B}[i])$
                  where  $sk := \mathsf{PRF}_{msk_1}(p^*, t, \text{“write-shuffle”})$,
                  $ek[i] := \mathsf{PRF}_{msk_1}(\mathsf{pos}', t, \text{“write-enc”})$
                   // $\mathsf{pos}'$ is $\mathbf{B}[i]$'s position after the shuffle
                   // $\mathsf{pos}'$ can be computed given $i'$
                **done**
$S_1 \to S_2$:  $\mathbf{B}'$
       $S_2$:   $\mathsf{BlocksAt}[p^*, \ell + 1] \leftarrow \mathbf{B}'$
 $S_1 \to C$:   “Done.”
 $S_2 \to C$:   “Done.”
        $C$:   Update local metadata appropriately.

**Case 2:** $\ell = L - 1$

 $C \to S_1$:  “Write partition”,  $p^*$,  $B$
        $C$:  $I \leftarrow \emptyset$
              For each level $i = 0$ to $L - 1$:
                   let $I_r := \{$positions of all unread real blocks$\}$,
                   let  $I_d := \{$positions of $2^i - |I_r|$ randomly
                               chosen unread dummy blocks$\}$,
                   $I \leftarrow I \cup (I_r \cup I_d)$
 $C \to S_1$:  $\mathsf{sorted}(I)$
      $S_1$ :  $\mathbf{B} \leftarrow \mathsf{BlocksAt}[I] \cup \{B\}$
       $S_1$:  $\forall i \in |\mathbf{B}|$ :  **do**
                   Let $i' = \mathsf{PRP}_{sk}(i)$,  $\mathbf{B}'[i'] := E_{ek[i]}(\mathbf{B}[i])$
                   where  $sk := \mathsf{PRF}_{msk_1}(p^*, t, \text{“write-shuffle”})$,
                   $ek[i] := \mathsf{PRF}_{msk_1}(\mathsf{pos}', t, \text{“write-enc”})$
                    // $\mathsf{pos}'$ is $\mathbf{B}[i]$'s position after the shuffle
                     // $\mathsf{pos}'$ can be computed given $i'$
              **done**
$S_1 \to S_2$:  $\mathbf{B}'$
       $S_2$:  $\mathsf{BlocksAt}[p^*, L - 1] \leftarrow \mathbf{B}'$
 $S_1 \to C$:  “Done.”
 $S_2 \to C$:  “Done.”
        $C$:  Update local metadata appropriately.
```

Figure 4: Partition read and write algorithms for the honest-but-curious model.

Whenever a cloud, say S_1, needs to shuffle a set of blocks, it generates a pseudo-random one-time shuffling key based on msk_1, the current partition, and its the time value.

Onion decryption and background onion removal. The client can recover all encryption keys and decrypt any onion-encrypted block. The details of this is deferred to the full online version [29] for our full construction — which is based on the basic construction described in this section, but augmented to provide security when one of the two clouds is malicious.

4. SECURITY AGAINST ONE MALICIOUS CLOUD

Informally, a malicious server can 1) corrupt data in storage; and 2) deviate from the prescribed protocol, particularly, not performing shuffling correctly.

Corrupted data blocks can be detected through standard techniques such as message authentication codes (MAC), as described in several earlier works [11, 31]. Therefore, we focus our attention on how to detect deviations from the prescribed protocol behavior.

High-level idea. The high-level idea is as follows. The client has two secret checksum functions σ_1 and σ_2. σ_1 is shared with cloud S_1, and σ_2 is shared with cloud S_2. *The client attaches encrypted and authenticated version of the two checksums, denoted $\tilde{\sigma}_1(B)$ and $\tilde{\sigma}_2(B)$ to each block B and stored on the servers along with the block.* The checksums are encrypted and authenticated with a private key that only the client knows.

The client treats the checksums the analogously to the way the clouds treats the blocks. For example, whenever a cloud permutes a set of blocks, the client will permute the corresponding checksums. Whenever a cloud adds an onion layer of encryption to a block, the client will compute the

S_1:	$\mathbf{B}' \leftarrow$ onion-encrypt and shuffle $\mathbf{B}$. *// Let π denote the shuffling permutation.*
$S_1 \to S_2$:	$\mathbf{B}'$
$S_1 \to C$:	$\{\tilde{\sigma}_1(B)$ and $\tilde{\sigma}_2(B) : \forall B \in \mathbf{B}\}$
$S_2 \to C$:	$\{\sigma_2(B')$: $\forall B' \in \mathbf{B}'\}$
C:	$\forall B \in \mathbf{B}$: $\sigma_1(B) :=$ Decrypt $\tilde{\sigma}_1(B)$ $\sigma_2(B) :=$ Decrypt $\tilde{\sigma}_2(B)$
C:	Verify that $g_{ek_i}(\sigma_2(B_i)) = \sigma_2(B'_{\pi(i)})$ for $0 \le i < \lvert\mathbf{B}\rvert = \lvert\mathbf{B}'\rvert$ where ek_i is the time and position dependent one-time encryption key (for block B_i known only to C and S_1).
C:	Compute $\sigma_1(B')$ from $\sigma_1(B)$ for all blocks.
C:	$\forall B' \in \mathbf{B}' : \tilde{\sigma}_1(B') :=$ Encrypt & authenticate $\sigma_1(B')$
$C \to S_2$:	$\{\tilde{\sigma}_1(B') : \forall B' \in \mathbf{B}'\}$

Figure 5: Verifying shuffling and onion-encryption performed by cloud S_1. The protocol is symmetric and the same verification can be done with for S_2 by swapping the roles of S_1 and S_2. The verification is bandwidth efficient in that the client only needs to transfer checksums and not the blocks themselves. Encryption and authentication of the checksums is performed using a secret key known only to the client.

checksum for the onion-encrypted block from its old checksum. After updating a checksum and before uploading it back to the clouds, the client always re-encrypts it and re-authenticates it using authenticated encryption.

Suppose Cloud S_1 onion encrypts and shuffles a set of blocks and sends them to S_2. The client can verify that S_1 did this correctly by asking S_2 to compute and send the checksums of the shuffled and onion-encrypted blocks it received from S_1. The client can then verify the checksums it received from S_2 against the ones it computed directly from the checksums of the old blocks (before shuffling and onion encryption).

Detailed algorithm. Figure 5 describes in more detail how the client can verify that a set of blocks have been correctly shuffled and onion-encrypted by cloud S_1. The same protocol can be run with S_1 and S_2 interchanged to verify shuffling and onion encryption performed by S_2. We prove the security of this verification algorithm in the full online version [29].

In order make the verification in Figure 5 possible, our checksum construction is designed to have the following properties.

Commutative checksum/encryption construction. The client can compute the new checksum of a block (after onion encryption by S_1) from an old checksum (before onion encryption by S_1), *without having to download the block itself.* In other words, for $j \in \{1, 2\}$, there is an efficiently computable function g_{ek} taking the encryption key ek as the key, such that

$$g_{ek}(\sigma_j(B)) = \sigma_j(E_{ek}(B)) \qquad (1)$$

Unforgeability of checksum function. For the above construction to be secure, the checksum function σ needs to satisfy the following unforgeability property: when the checksum function σ is kept secret from an adversary (i.e., one of the clouds), and the adversary has *not* observed any input-output pairs, then, the adversary cannot find a pair $B \neq B'$ such that $\sigma(B) = \sigma(B')$ except with negligible probability. Note that our checksum function is *not* a hash function in the traditional sense, since for a traditional hash function, the description of the function is public.

Intuitively, this property ensures that S_1 cannot deviate from the shuffling protocol: suppose that after a shuffle from S_1 to S_2, the correct block at permuted index i' should be B'. If S_1 sent to S_2 some other $B^* \neq B'$ for index i', then $\sigma_2(B') \neq \sigma_2(B^*)$ except with negligible probability. Hence, the client can immediately detect such deviation.

This property can be formalized as below, and is used as a building block in our full proof in the full online version [29].

Definition 1 (Unforgeability). *We say that a checksum function family Σ has our special collision resistance property if for all adversaries $\mathcal{A}$ (even when $\mathcal{A}$ is computationally unbounded),*

$$\Pr\left[\begin{array}{l} \textit{Pick random } \sigma \textit{ from } \Sigma; \\ (B_0, B_1) \leftarrow \mathcal{A}(1^\lambda) \end{array} : \begin{array}{l} \sigma(B_0) = \sigma(B_1) \\ \wedge\ B_0 \neq B_1 \end{array}\right] = \mathsf{negl}(\lambda)$$

4.1 Commutative Checksum-Encryption Construction

Counter mode of AES encryption. We use counter mode of AES encryption. To encrypt a block $B \in \{0,1\}^\beta$, with a *one-time* key $ek \in \{0,1\}^\kappa$ where κ is 128 bits or 256 bits, compute the ciphertext $R \oplus B$ where R is defined as

$$R := \textit{prefix}\,(\mathsf{AES}_{ek}(0)||\mathsf{AES}_{ek}(1)||\mathsf{AES}_{ek}(2)\ldots) \qquad (2)$$

In the above *prefix* denotes the function that takes the first β bits of the string input. This is a secure one-time encryption scheme, since *each ek is used to encrypt only one message.*

Checksum construction. Let $\sigma : \{0,1\}^\beta \to \{0,1\}^\lambda$ denote our specially constructed checksum function, where β is the block size, and λ is the security parameter with a typical choice of 128. Our checksum function σ is defined by a $\lambda \times \beta$ matrix $M \in \{0,1\}^{\lambda\beta}$, where λ is the number of rows, and β is the number of columns. Each entry in the matrix is a bit picked at random from $\{0,1\}$. Checksuming a block is simply a matrix multiplication operation:

$$\sigma(B) := M \cdot B \pmod 2$$

Commutative property of checksum-encryption combination. Given an encryption key ek, and a previous checksum σ, we define a verification function g taking ek and σ as inputs:

$$g_{ek}(\sigma) := \sigma \oplus (M \cdot R) \qquad (3)$$

where the vector R is defined as in Equation (2) — taking ek as input.

Lemma 1. *The above encryption and checksum combination satisfy Equation (1) for any $ek \in \{0,1\}^\kappa$, and any block $B \in \{0,1\}^\beta$,*

Proof. Note that XOR denoted $\oplus$ is simply addition mod 2. Let $R \in \{0,1\}^\beta$ be a function of ek as defined by Equation (2). Therefore,

$$g_{ek}(\sigma(B)) = \sigma(B) \oplus (M \cdot R)$$
$$= (M \cdot B) \oplus (M \cdot R) = M \cdot (B \oplus R) = \sigma(E_{ek}(B))$$

$\square$

Lemma 2 (Security of checksum function). *Our checksum function is secure by Definition 1.*

The proof of this lemma is deferred to the full online version [29].

Authenticated encryption of checksums. As mentioned earlier, our checksum function is secure only when an adversary has not seen any checksum value. Furthermore, we also need to ensure that a malicious cloud cannot modify the checksum and claim a different checksum. We therefore need to ensure the confidentiality and authenticity (implies freshness) of the checksums. To do this, the client will rely on authenticated encryption to encrypt these checksums before attaching them to the blocks and uploading them to a cloud. To ensure freshness, the authenticated encryption incorporate the *time* of write and the *position* as well. One way to do this is the following: for each block shuffled at time t and shuffled to position pos on cloud S_j, generate a one-time authenticated encryption key ak based off a master secret key ck known only to the client:

$$ak := \mathsf{PRF}_{ck}(\mathsf{pos}, t, j, \text{“AE-Write-Shuffle”})$$
$$\tilde{\sigma} := \mathsf{AE}_{ak}(\sigma)$$

where AE is an authenticated encryption scheme. For shuffles that occur in the **ReadPartition** algorithm (see Figure 4), a different tag “AE-Read-Shuffle” may be used.

Optimization. In description so far, S_2 sends the client a new checksum $\sigma_2(B')$ for each block B' after shuffling, and the client verifies them one-by-one. A simple optimization is for S_2 to use a collision-resistant hash function (e.g., SHA-256), and hash all of the $\sigma_2(B)$'s, and send a single hash to the client C — who can then verify the hash by reconstructing all the $\sigma_2(B)$'s.

4.2 Verifying Fetched Blocks

Whenever the client reads any data block using the **ReadPartition** algorithm, it needs to verify the authenticity and freshness of the fetched block. Our encrypted and authenticated checksums $\tilde{\sigma}_1(B)$ and $\tilde{\sigma}_2(B)$ allow the client to verify the authenticity and freshness for the block B. Particularly, the freshness property is due to the fact the authenticated encryption uses a time and position dependent key.

In the **ReadPartition** algorithm, we simply use the aforementioned algorithm to verify the shuffles associated with read operations. Finally, after fetching a block B, the client verifies its authenticity and freshness using $\tilde{\sigma}_1(B)$ or $\tilde{\sigma}_2(B)$.

5. EXPERIMENTAL RESULTS

5.1 Overview and Experimental Setup

We have implemented a full-fledged multi-cloud ORAM implementation, consisting of 12,000 lines of code. Our implementation uses hardware-accelerated AES-NI whenever available (it is available on most modern Intel processors).

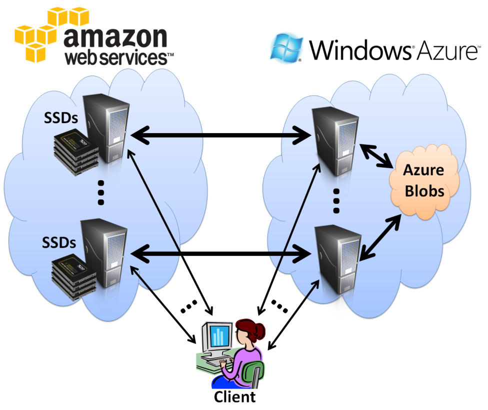


Figure 6: Our deployment. The workload is distributed across multiple servers per cloud communicating in pairs. We use this deployment for the experiments.

Our ORAM implementation uses asynchronous I/O operations to increase throughput. As Stefanov and Shi point out in their recent work [30], in an asynchronous ORAM, to avoid information leakage through timing of I/O events, we need to make sure that the scheduling module does not use any information related to the logical access pattern. We therefore follow this guideline in our multi-cloud ORAM design, to ensure security in the asynchronous ORAM model.

Deployment. We deployed our multi-cloud ORAM implementation running on top of two real-world major cloud providers: Amazon Web Services (AWS) [1] and Microsoft Azure [4]. We rented up to 5 servers per cloud. The AWS servers were High I/O Quadruple Extra Large Instances, each with 2 SSD-based volumes (1 TB each). The Azure servers were Extra Large instances, with the Azure blob storage as the storage backend (SSDs were not available on Azure). Figure 6 illustrates our deployment.

Although these VM instances have large memory and fast CPUs, we chose them only because they are provisioned by the cloud providers with higher disk I/O and network I/O than other instances. As explained in Section 5.4 and Section 5.5, CPU and memory were never the bottlenecks in our experiments.

Latency. Unless otherwise specified, we simulate a 50ms (round-trip) latency between the client and the closest cloud by asynchronously delaying the client's requests and responses. Since our two clouds are in fact two different clouds (Amazon and Azure), we do not simulate additional latency between the clouds.

Scaling. We scale our experiments up to 5 servers per cloud, because our experiments suggest that by this time, the client-cloud bandwidth would already have been saturated in most typical settings – hence, further scaling up inside the clouds would not have increased the overall ORAM throughput. We consider a single client as in most existing ORAM work.

Warming up. All of our experimental results represent the performance of a warmed-up ORAM with N blocks. We use the same technique for warming-up the ORAM as in previous work [30, 31]. As explained in the full online version [29], the client initializes the internal data structures of our construction so that the ORAM immediately starts-off in a warmed-up state (i.e., with multiple filled levels in each partition) without having to pre-initialize or zero-out the server-side storage. This ensures that our performance measurements are steady-state measurements, not burst or best-case performance.

Access pattern. Because our ORAM construction protects the privacy of the access pattern, the access pattern will have no distinguishable effect on the performance of the clouds. However, the access pattern can affect the size of the client's internal data structures (up to a certain point). In order to fairly evaluate our construction, we used a round-robin access pattern as in previous work [30, 31], which turns out to maximize the client storage by reducing the possibility of reusing data already stored by the client.

For ORAM, reads and writes are also indistinguishable. In our construction, they each involve invoking **ReadPartition** and then **WritePartition**, except that during write operation, the client locally updates the block's value in between the **ReadPartition** and **WritePartition** operations. Our access pattern hence consisted of writes.

System load. All of our experiments represent the performance of our system under full load. In other words, data is continuously being written to the ORAM as fast as the ORAM can handle it. For each trial of each data point, we started measuring the performance after 200 MB of data had been written to the ORAM and we stopped measuring after 400 MB of data had been written (committed to disk by the cloud). Although the client memory is up to 1.5 GB for a 1 TB ORAM, most of it is used for the position map, and semaphores in our implementation ensure that only at most 70 MB of it is used to temporarily store data blocks before they are evicted to the clouds. Therefore by starting our measurements after 200 MB of data is written, we eliminate the effect of data buffering by the client.

Number of trails. Except for Figure 15, which provides exact values calculated for our construction, each data point in each figure represents the average over 20 trials. The error bars represent one standard deviation.

5.2 Results: Single Server Per Cloud

We first experimented with a single server per cloud, and report our experimental results below.

Throughput to a large extent depends on the bandwidth of the bottleneck resource. If the client-cloud link is the bottleneck, then our throughput would be roughly $BW/2.6$ where BW denotes the available bandwidth on the client-cloud link. We refer to the number 2.6 as the client-cloud bandwidth cost, i.e., to access a block, roughly 2.6 blocks need to be transferred across the client-cloud link. A breakdown of the client-cloud bandwidth cost is shown in Figure 12.

In Figures 7, we measure maximum ORAM throughput that our multi-cloud system can handle with with 1 server per cloud when the client-to-cloud bandwidth is ample. A 300 GB ORAM of 4 KB blocks with a client over a 50ms latency connection to the clouds can handle up to about 0.8 to 1.0 MB/s throughput. If the block size is increased to 8 KB or 16 KB, the ORAM can sustain 1.2 to 1.6 MB/s of throughput. The graph shows that for ORAMs of sizes between 50 and 300 GB, the throughput is about the same. Theoretically it should decrease by about 15% because an ORAM of size 300 GB has about 14 levels per partition whereas an ORAM with size 50 GB has about 12 levels per partition and the cost I/O cost of the ORAM is proportional to the number of levels in the partitions. This slight decrease in performance is somewhat noticeable but it's mostly masked by variance due to external factors such as fluctuation in cloud network performance.

Figures 8,9, and 10 show the response time of our system under different parameterizations and the effect that the client-to-cloud latency has on the system throughput and response time. In Figure 10 the dotted line is the ideal $y = x$ curve, and represents the case when the request can be handled in exactly the client-to-cloud latency. The response-time of our system is about 200ms to 300ms higher than the client-to-cloud latency due to the cloud-to-cloud latency and the network congestion created by the fact that our system is under full load.

In these figures, the actual client-cloud bandwidth consumption is about 2.6 times the ORAM throughput.

5.3 Scaling to Multiple Servers per Cloud

Figure 11 shows the scaling effect when we use multiple servers per cloud as described in the full online version [29]. Each server handles roughly 300GB of ORAM capacity, and the load distribution scheme is described in the full online version [29]. The total ORAM capacity is therefore 300GB times the number of servers per cloud. We can see that the throughput roughly scales linearly as the number of servers grow (when the client-cloud link is not the bottleneck). The variations are due mostly to varying network performance between specific servers in the clouds. Our experiments also suggest that the response time grows very slightly as the number of servers per cloud grows (Figure 15).

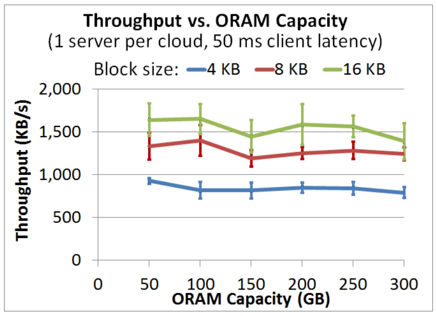
We did not further scale up our experiments beyond 5 servers per cloud, since in the real-world settings we consider, the client-cloud bandwidth would already have been saturated at this scale.

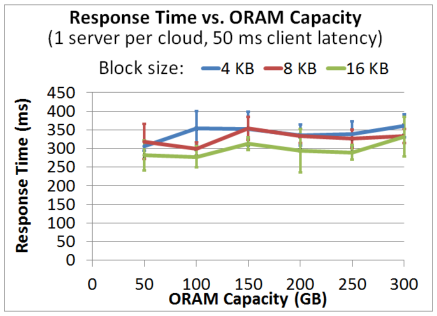
5.4 Bottleneck Analysis

Client-cloud link is the bottleneck. As mentioned earlier, if the client-cloud link is the bottleneck, the ORAM throughput would just be $BW/2.6$, where BW is the client-cloud bandwidth.

Storage performance is the bottleneck. When the client-cloud link has ample bandwidth, the next immediate bottleneck is storage performance. In our deployment, since Azure did not offer SSD-capable instances, Azure's blob storage becomes the bottleneck. Figures 7, 8, 9, 10, and 11, reported above focus on this case.

Inter-cloud networking is the bottleneck. Figures 14 and 15 report the throughput and response time we could potentially obtain had the storage performance not been the bottleneck – e.g., if Azure could provide SSD as their storage; or if in the future, RAM-based persistent storage becomes available (e.g., RAMClouds [24]). In our deployment scenario, the cloud-cloud network bandwidth varied from 30–60MB/s between a pair of servers, which is slower than typical SSD performance (about 150MB/s). Therefore, to extrapolate the performance we could obtain had Azure provided SSD-based storage, in Figures 14 and 15 we emulate the storage back-end using `/dev/zero` (or its equivalent on Windows) – such that the cloud-cloud network is the bottleneck.





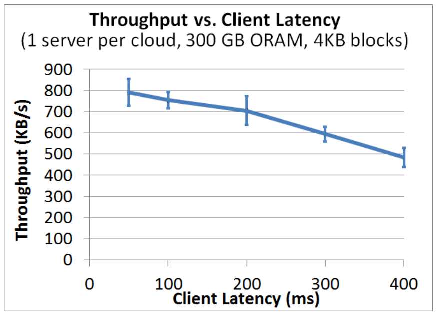
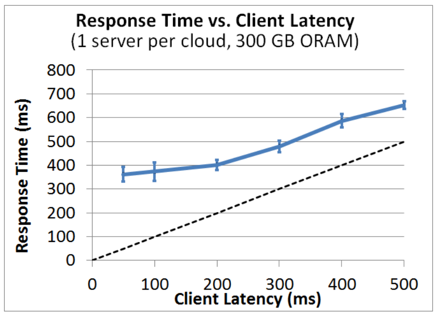
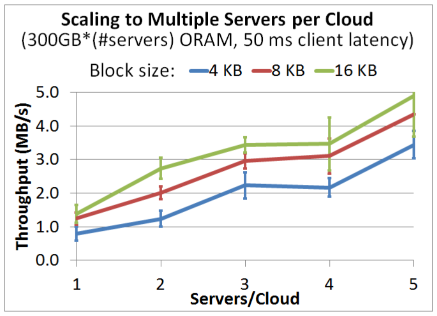


Figure 7: ORAM throughput vs. capacity. With 1 server per cloud, and a simulated 50ms latency client-cloud network link.

Figure 8: ORAM response time vs. capacity. With 1 server per cloud, and a simulated 50ms latency client-cloud network link.

Figure 9: Effect of client-cloud network latency on throughput. With 1 server per cloud, and an ORAM of 300GB capacity.





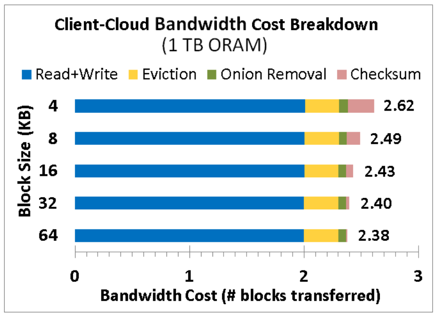
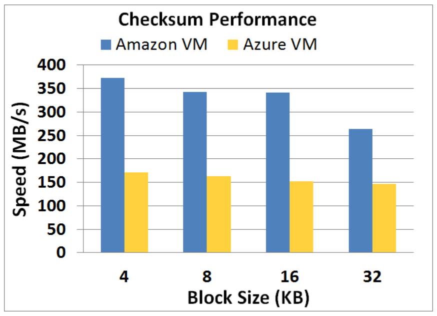
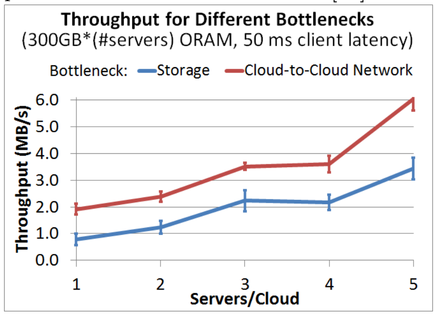


Figure 10: Effect of client-cloud network latency on response time. With 1 server per cloud, and an ORAM of 300GB capacity. The dotted line represents the network roundtrip latency – this is the lower bound on response time.

Figure 11: Scaling to multiple servers per cloud: Throughput. With a simulated 50ms latency client-cloud network link. The total ORAM capacity is 300GB times the number of servers. The workload distribution scheme across servers is explained in the full online version [29].

Figure 12: Breakdown of client-cloud bandwidth cost. Measured between the client and all clouds combined for a single ORAM read/write. The number of servers per cloud does not affect the client-cloud bandwidth cost.





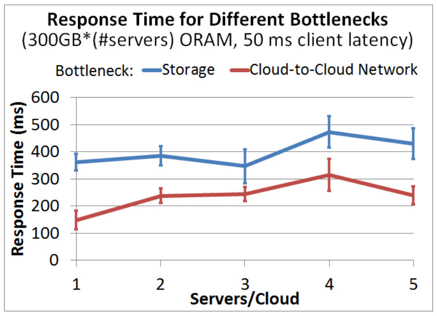


Figure 13: Microbenchmarks for checksum computation. Rate at which each VM in our experiments is able to perform the commutative checksum computation (parallelizing across all cores). Checksum performance slightly decreases with the block size due to a decrease in memory locality as the checksum matrix increases proportionally to the block size.

Figure 14: Potential improvement in throughput if storage were fast enough (for 4 KB blocks). The "Storage" curve represents the real-world case where the storage performance is the bottleneck. The "Cloud-cloud network" curve emulates the storage with `/dev/zero` such that the cloud-cloud network link is the bottleneck.

Figure 15: Potential improvement in response time if storage were fast enough (for 4 KB blocks). The "Storage" curve represents the real-world case where the storage performance is the bottleneck. The "Cloud-cloud network" curve emulates the storage with `/dev/zero` such that the cloud-cloud network link is the bottleneck.

5.5 Breakdown of Costs

Bandwidth baseline. Our baseline for bandwidth is a simple cloud storage system that just reads and writes unencrypted blocks to storage. For example, reading or writing a 4 KB block from the baseline system will incur 4 KB of bandwidth. When we say that there is kX ORAM bandwidth cost, we mean that it takes k times as much bandwidth to perform an ORAM read or a write operation.

Client-to-cloud bandwidth. Our construction achieves about 2.6X bandwidth cost which means it takes about 10,647 bytes to read or write 4096 bytes from our multi-cloud ORAM. Figure 12 shows the breakdown of the client-cloud bandwidth cost. Particularly, about 2X (out of a total of ~ 2.6X) of the bandwidth cost is due to hiding whether the block is being read or written (i.e., fetching one block of data with 1X cost and writing one block of data with 1X cost).

For the SSS partitioning framework (Section 2), we use a background eviction rate at 0.3 times rate of data access (in addition to the piggybacked write-back with each operation). Therefore, background evictions account for 0.3X client-cloud bandwidth cost. Due to space limitations, the reader is encouraged to refer to [31] for details about the background eviction process.

The remaining 0.3X or so bandwidth cost is due to the background onion removal process and the transfer of (encrypted and authenticated) checksums between the client and cloud – as mentioned in Section 4, these checksums are necessary to ensure security against one malicious cloud. The checksum size is independent of the block size therefore when the block size increases, the checksum fraction of the bandwidth cost decreases.

Cloud-to-cloud bandwidth. The bandwidth cost of the entire system, which mostly consists of the cloud-to-cloud bandwidth is a constant factor smaller than that for the best known single-cloud construction [30] using the same amount of client memory. The intuition behind why that is the case is as follows. In the single-cloud ORAM by Stefanov *et al.* [30], when blocks need to be shuffled, the client downloads the blocks, shuffles them locally, and uploads them back to the cloud in order to avoid increasing the the client's storage. However, in our multi-cloud construction, one of the clouds just shuffles the blocks and sends them to the other cloud and the other cloud simply stores them and does not need to send the blocks back to the first cloud.

Cryptographic microbenchmarks. We report microbenchmarks for our new checksum function described in Section 4.1. Figure 13 shows that our checksum function can be computed at roughly 250-350 MB/s on our Amazon VM, and at 150 MB/s on our Azure VM. Using hardware-accelerated AES-NI in modern processors, decryption can be performed at 2.5 GB/s per onion layer.

Client computation. The client computation for a 1 TB ORAM mainly consists of: (1) encrypting and computing the initial checksum of a block when it is written to the ORAM, (2) encrypting about 15 checksums (16 bytes each) per block written using the commutative checksum-encryption technique described in Section 4.1, and (3) decrypting using AES about 5-15 onion layers of a fetched block that were added by the clouds during shuffling.

Even without hardware AES available on the client, a modern laptop or desktop computer can easily perform these cryptographic operations to sustain several megabytes per second of ORAM throughput, and most likely saturate its Internet connection (depending on the connection available to the client).

Server computation. In order for a server to sustain about 1 MB/s per sever of bandwidth for a 1 TB ORAM, the the server needs to be able to sustain about 30 MB/s of AES computation and 30 MB/s checksum computation over 4 KB blocks. As we just mentioned, our cryptographic microbenchmarks show that the Amazon and Azure servers we rented can sustain at least 5 to 10 times that rate for checksums and over 80 times that rate for AES.

Client memory and storage. Our client-side storage (memory) is less than 1.5 GB for an ORAM of up to 1 TB capacity (i.e., less than 0.15% of the entire ORAM capacity).

Monetary cost. The monetary cost of our system depends on several factors such as (1) the desired throughput and response time, (2) the size of the ORAM, (3) idle time, and (4) other factors such as geographic location. At the time the experiment was run, under full load, it's cost was about \$3.10/hour + \$2.50/GB data transfer for 1-server per cloud. By using Amazon S3 and cheaper VMs instead of high I/O VM instances, the hourly cost can likely be reduced significantly.

6. RELATED WORK

Oblivious RAM was first proposed by Goldreich and Ostrovsky [11] in the context of protecting software from piracy. They propose a seminal hierarchical construction with $O((\log N)^3)$ amortized bandwidth cost, where N denotes the storage capacity of the ORAM. Since then, a line of research in the theory community has been dedicated to ORAM [7, 9–11, 13–16, 19, 22, 23, 25, 32–34].

Several works have suggested the use of ORAM in cloud computing applications [13, 15, 16, 30, 32, 34, 35]. Williams, Sion *et al.* have significantly bridged the theory and practice of ORAM [32, 34, 35]. Goodrich, Mitzenmacher, Ohrimenko, Tamassia *et al.* [15, 16] have also made significant contributions in this space. Backes *et al.* [6] use a combination of the binary-tree ORAM [27] and trusted hardware to build privacy-preserving behavioral advertising applications.

Several recent efforts have made further progress towards making ORAM practical, including PrivateFS [35], Shroud [20], and ObliviStore [30]. PrivateFS and ObliviStore show how to build parallel or asynchronous ORAM schemes that achieve throughput in the range of hundreds of kilobytes per second on a single disk-bound server — assuming the client-cloud bandwidth is not the bottleneck. Unless trusted hardware is deployed in the cloud, these schemes result in 20X-35X client-cloud bandwidth cost, and is unsuitable for bandwidth constrained clients. Our multi-cloud ORAM is able to achieve a lower overall bandwidth cost than both ObliviStore [30] and PrivateFS [35], and an even lower client-cloud bandwidth cost. Shroud shows how to implement ORAM with trusted co-processors such as IBM 4764 [2], and scale it up in a distributed data center setting, but its performance is severely limited by the trusted co-processors. In comparison, our goal is to provide an implementation that is readily deployable today (i.e., not relying on trusted hardware), and that addresses the client-cloud bandwidth bottleneck.

As mentioned earlier, Lu and Ostrovsky study multi-server oblivious RAM from a theoretic perspective [21]. They show that if there exists two non-colluding servers, the total bandwidth cost of oblivious RAM (including messages sent between the client and the servers, as well as amongst the servers themselves) may be reduced to $O(\log N)$ under the constant client local storage setting. Their work is mostly theory-minded, and results in a high client-server bandwidth in practice. Lu and Ostrovsky's non-collusion model is stronger than ours, since their two non-colluding servers do not communicate with each other.

7. CONCLUSION

In this paper, we described a practical two-cloud Oblivious RAM protocol that *reduces the client-server bandwidth cost to about* 2.6 *times that of simply reading or writing the block from non-oblivious cloud storage.* In comparison, for the same amount of client memory available to the client, the best-known existing (single-cloud) ORAM constructions have a bandwidth cost of about 20X–35X.

We proposed a novel commutative checksum-encryption construction that allows our multi-cloud ORAM protocol to to efficiently protect the privacy of the access pattern against one malicious cloud (without necessarily knowing which one).

We implemented a complete end-to-end systemdeployed on multiple servers across two clouds (Amazon EC2 and Windows Azure) and demonstrated that it can scale and achieve several megabytes of throughput, saturating the available bandwidth of most typical client Internet connections.

Acknowledgments

This work is partially supported[2] by an NSF Graduate Research Fellowship under Grant No. DGE-0946797, by a DoD National Defense Science and Engineering Graduate Fellowship, NSF grant CNS-1314857, a Google Research Award, and a grant from Amazon Web Services. We would like to thank Matthew Green for helpful discussions and the anonymous reviewers for their insightful feedback.

8. REFERENCES

[1] Amazon web services. `http://aws.amazon.com/`.

[2] IBM 4764 PCI-X cryptographic coprocessor (PCIXCC). `http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`.

[3] Trusted computing group. `http://www.trustedcomputinggroup.org/`.

[4] Windows azure. `http://www.windowsazure.com/`.

[5] D. Asonov and J.-C. Freytag. Almost optimal private information retrieval. In *PET*, 2003.

[6] M. Backes, A. Kate, M. Maffe, and K. Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *S & P*, 2012.

[7] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, 2011.

[8] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *CCSW*, 2009.

[9] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.

[10] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[11] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

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

[13] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[14] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *ACM Cloud Computing Security Workshop (CCSW)*, 2011.

[15] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *CODASPY*, 2012.

[16] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[17] A. Iliev and S. W. Smith. Protecting client privacy with trusted computing at the server. *IEEE Security and Privacy*, 3(2):20–28, Mar. 2005.

[18] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[19] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[20] J. R. Lorch, J. W. Mickens, B. Parno, M. Raykova, and J. Schiffman. Shroud: Enabling private access to large-scale data in the data center. In *FAST*, 2013.

[21] S. Lu and R. Ostrovsky. Distributed oblivious ram for secure two-party computation. Cryptology ePrint Archive, Report 2011/384, 2011. `http://eprint.iacr.org/`.

[22] R. Ostrovsky. Efficient computation on oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1990.

[23] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[24] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.

[25] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[26] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *CCSW*, pages 43–46, 2010.

[27] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[28] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Syst. J.*, 40(3):683–695, Mar. 2001.

[29] E. Stefanov and E. Shi. Multi-cloud oblivious storage. Technical report.

[30] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy*, 2013.

[31] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[32] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[33] P. Williams and R. Sion. Round-optimal access privacy on outsourced storage. In *CCS*, 2012.

[34] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.

[35] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.

[2]Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.