2013 IEEE Symposium on Security and Privacy

ObliviStore: High Performance Oblivious Cloud Storage

Emil Stefanov
University of California, Berkeley
emil@cs.berkeley.edu

Elaine Shi
University of Maryland, College Park
elaine@cs.umd.edu

Abstract. We design and build ObliviStore, a high performance, distributed ORAM-based cloud data store secure in the malicious model. To the best of our knowledge, ObliviStore is the fastest ORAM implementation known to date, and is faster by 10X or more in comparison with the best known ORAM implementation. ObliviStore achieves high throughput by making I/O operations asynchronous. Asynchrony introduces security challenges, i.e., we must prevent information leakage not only through access patterns, but also through timing of I/O events. We propose various practical optimizations which are key to achieving high performance, as well as techniques for a data center to dynamically scale up a distributed ORAM. We show that with 11 trusted machines (each with a modern CPU), and 20 Solid State Drives, ObliviStore achieves a throughput of 31.5MB/s with a block size of 4KB.

I. Introduction

Cloud computing provides economies of scale for implementing a broad range of online services. However, due to concerns over data privacy, "many potential cloud users have yet to join the cloud, and many are for the most part only putting only their less sensitive data in the cloud" [10]. It is well-known that encryption alone is not sufficient for ensuring data privacy, since data access patterns can also leak a considerable amount of sensitive information. For example, Islam *et. al.* demonstrate that access patterns can leak (through statistical inference) up to 80% of the search queries made to an encrypted email repository [21].

Oblivious RAM (or ORAM for short) [9, 11, 13–16, 18, 23, 28, 29, 31, 43, 46], originally proposed by Goldreich and Ostrovsky [14], is a cryptographic construction that allows a client to access encrypted data residing on an untrusted storage server, while completely hiding the access patterns to storage. Particularly, the sequence of physical addresses accessed is independent of the actual data that the user is accessing. To achieve this, existing ORAM constructions [9, 11, 13–16, 18, 23, 28, 29, 31, 43, 46] continuously re-encrypt and and reshuffle data blocks on the storage server, to cryptographically conceal the logical access pattern.

Aside from storage outsourcing applications, ORAM (in combination with trusted hardware in the cloud) has also been proposed to protect user privacy in a broad range of online services such as behavioral advertising, location and map services, web search, and so on [8, 25].

While the idea of relying on trusted hardware and oblivious RAM to enable access privacy in cloud services is promising, for such an approach to become practical, a key challenge is the practical efficiency of ORAM. ORAM was initially proposed and studied mostly as a theoretic concept. However, several recent works demonstrated the potential of making ORAM practical in real-world scenarios [25, 40, 46, 47].

A. Our Contributions

We design and build ObliviStore, an efficient ORAM-based cloud data store, securing data and access patterns against adversaries in the *malicious* model. To the best of our knowledge, ObliviStore is the fastest ORAM implementation that has ever been built.

Our evaluation suggests that in a single client/server setting with 7 rotational hard disk drives (HDDs), ObliviStore is *an order of magnitude faster* than the independent work PrivateFS by Williams *et. al.* [47] – with parameters chosen to best replicate their experimental setup (Section VII-E).

As solid-state drive (SSD) prices drop faster than HDDs [30], cloud providers and data centers embrace SSD adoption [4]. In addition to HDDs, we also evaluate ObliviStore with SSDs in a *distributed* setting on Amazon EC2. With 11 trusted nodes (each with a modern CPU), we achieve a throughput of $\mathbf{31.5MB/s}$ with a block size of 4KB.

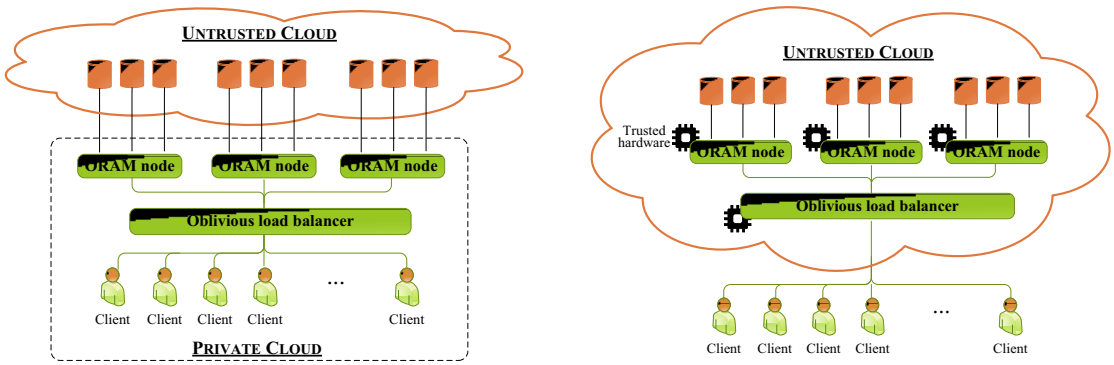
Our technical contributions include the following:

Making ORAM operations asynchronous. We propose novel techniques for making the SSS ORAM [40] asynchronous and parallel. We chose the SSS ORAM since it is one of the most bandwidth efficient ORAM constructions known to date. Due to ORAM's stringent security requirements, making ORAM operations asynchronous poses unique challenges. We must prevent information leakage not only through access patterns as in traditional synchronous ORAM, but also through the timing of I/O events. To address this issue, we are the first to formally define the notion of *oblivious scheduling*. We prove that our construction satisfies the oblivious scheduling requirement. Particularly, our ORAM scheduler relies on semaphores for scheduling. To satisfy the oblivious scheduling requirement, operations on semaphores (e.g., incrementing, decrementing) must depend only on information observable by an adversary who is not aware of the data request sequence.

Distributed ORAM. Typical cloud service providers have a distributed storage backend. We show how to adapt our ORAM construction for a distributed setting.

Note that naive methods of partitioning and distributing an ORAM may violate security. For example, as pointed out in [40], even if each block is assigned to a random

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.25



(a) Scenario 1: private + public cloud.

(b) Scenario 2: trusted hardware in the cloud.

Figure 1: Architecture and deployment scenarios.

partition when written, accessing the same block twice in a row (read after write) can leak sensitive information. Our distributed ORAM construction applies the SSS partitioning framework [40] twice to achieve secure partitioning of an ORAM across multiple servers.

We also propose a novel algorithm for securely scaling up a distributed ORAM at run-time. Our techniques allow additions of new processors and storage to an existing distributed ORAM without causing service interruption. As mentioned in Section VI, naive techniques for supporting dynamic node joins can easily break security. Non-trivial techniques are therefore required to securely handle dynamic node joins.

Practical optimizations. ObliviStore is designed to take into account many practical considerations (see full version [39]). For example, we use batch shuffling to boost the parallelism of the construction (Section V-C). We reorder and coalesce asynchronous I/O requests to storage to optimize the number of seeks on rotational drives.). We achieve parallelism through asynchronous operations with callbacks (rather than using more threads) to reduce thread scheduling contention. Read accesses have higher scheduling priority to minimize blocking on shuffling I/O's, and hence result in a lower overall response time, yet we make sure that the shuffling always keeps up with the accesses (Section V-D).

In ObliviStore, our oblivious load balancer stores about 4 bytes of metadata per data block. While the metadata size is linear in theory, its practice size is typically comparable to or smaller than storing $O(\sqrt{N})$ data blocks. For theoretic interest, with suitable modifications to the scheme, it is possible to achieve sublinear client storage by recursively outsourcing the metadata to the untrusted storage as well [25, 35, 40]. In practice, however, the recursion depth is typically 1 to 3 (see [25, 35]) — we use a value of 1, i.e., no recursion.

II. Architecture and Trust Model

Abstractly, all ORAM schemes assume a trusted client, and an untrusted storage provider. In our distributed ORAM, the trusted client consists of an *oblivious load balancer* and multiple *ORAM nodes* – we will explain the role of each in detail in Section VI. In practice, this means that we need to trust the part of the software implementing the oblivious load balancer and ORAM nodes. However, the rest of the system need not be trusted – specifically, we do not trust the network, the storage arrays, or the remainder of the software stack (other than the part that implements the oblivious load balancer and ORAM node algorithms).

ObliviStore is designed with two primary deployment scenarios in mind (Figure 1). The 1st scenario (hybrid cloud) is immediately deployable today, and PrivateFS [47] also considers a similar scenario. While the 2nd scenario (trusted hardware in the cloud) may not be immediately practical today, we envision it as a promising direction for building future privacy-preserving cloud services.

Hybrid cloud. One deployment scenario is corporate storage outsourcing. Suppose a company or government agency would like to outsource or backup its databases or file systems to untrusted cloud storage providers. In these cases, they may wish to separate the trusted components from the untrusted components, and host the trusted components in a private cloud in house, while outsourcing the untrusted storage to remote cloud providers. For example, Zhang *et. al.* [48] and others [41] describe such a hybrid cloud scenario in their paper. With ObliviStore, the oblivious load balancer and the ORAM nodes would reside in house, while the storage is provided by untrusted cloud providers. This scenario is also similar to that considered by Williams *et. al.* in their PrivateFS system [47].

Trusted hardware in the cloud. We envision a second deployment strategy as a promising direction to build a next generation of privacy-preserving cloud services.

ObliviAd [8] and Shroud [25] consider a similar scenario.

In various online services such as behavioral advertising and web search, access patterns reveal a great deal of sensitive information. For example, retrieving information about a certain drug can reveal a users medical condition; and retrieving information about a restaurant in New York can reveal the user's current location.

Several prior works [7, 8, 20, 25, 37, 43] have outlined the vision using trusted hardware [2, 5] to establish a "trust anchor" [34] in the cloud, which in turn relies on Oblivious RAM to retrieve data from untrusted storage while providing access privacy. For example, in S & P'12, Backes *et. al.* [8], propose to use Oblivious RAM in combination with trusted hardware, to ensure access privacy in online behavioral advertising. We can rely on Trusted Platform Modules (TPMs) [5, 26, 27] or secure co-processors [36, 38] to establish a Trusted Computing Base (TCB) at the cloud service provider. To achieve scalability, a distributed TCB is needed, and can be established through techniques such as in Excalibur [33].

In this scenario, our ORAM load balancer and ORAM node algorithms will be implemented as part of the distributed TCB, and will be in charge of encryption and privatizing access patterns. Other than the TCB, the remainder of the software stack on the cloud is untrusted. Existing work has also discussed how minimize the TCB to reduce the attack surface, and in some cases make it amenable to formal verification [22, 24, 42].

Using TPMs and Trusted Computing, we expect the distributed ORAM performance to be similar to the evaluations shown in this paper, since Trusted Execution imposes relatively small computational overhead. Moreover, this work shows that computation is not the bottleneck for ObliviStore when implemented on modern processors. On the other hand, off-the-shelf secure co-processors such as IBM 4768 may offer the additional benefit of physical security – but they are constrained (e.g., in terms of chip I/O, computation power, and memory) and would thus pose a bottleneck for an ORAM implementation, as demonstrated by Lorch *et. al.* [25]. However, it is conceivable that high performance secure co-processors suitable for ORAM can be built [12].

III. Preliminaries

A. Partitioning Framework

Stefanov, Shi, and Song propose a new paradigm for constructing practical ORAM schemes [40], consisting of two main techniques, *partitioning* and *eviction*. Through partitioning, they divide a bigger ORAM instance into multiple smaller ORAM instances. Let N denote the total ORAM capacity. The ORAM server storage is divided into $O(\sqrt{N})$ partitions, each with capacity $O(\sqrt{N})$.

At any point of time, a block resides in a random partition. The client stores a local *position map* to keep track of which partition each block resides in. To access a block

The partitioning framework [40]

// Divide the ORAM into $\sqrt{N}$ partitions each of capacity $\sqrt{N}$.

$\mathbf{Read}(\mathsf{blockid})$:

- Look up position map and determine that blockid is assigned to partition p.
- If blockid is not found in eviction caches:
  - $\mathbf{ReadPartition}(p, \mathsf{blockid})$

  Else if blockid is found in local eviction caches:
  - $\mathbf{ReadPartition}(p, \perp)$ `//read dummy`
- Pick a random partition p', add the block identified by blockid to the eviction caches, and logically assign blockid to partition p'.
- Call $\mathbf{Evict}$ ν times where $\nu > 1$ is the eviction rate.

$\mathbf{Evict}$:

- Pick a random partition p.
- If a block exists in the eviction cache assigned to partition p, write it back to partition p of the server.
- Else, write a dummy block to partition p of the server.

Figure 2: **The partitioning framework [40]**. The $\mathbf{Write}(\mathsf{blockid}, \mathsf{block})$ operation is omitted, since it is similar to $\mathbf{Read}(\mathsf{blockid})$, except that the block written to the eviction cache is replaced with the new block.

B, the client first looks up this position map to determine the partition id p; then the client makes an ORAM call to partition p and looks up block B. 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*.

A background eviction process evicts blocks from the eviction cache back to the server in an oblivious manner. One possible eviction strategy is random eviction: with every data access, randomly select 2 partitions for eviction. If there exists a block in the eviction cache that is assigned to the chosen partition, evict a real block; otherwise, evict a dummy block to prevent information leakage.

The basic SSS ORAM algorithm is described in Figure 2. Stefanov *et. al.* prove that the client's eviction cache load is bounded by $O(\sqrt{N})$ with high probability. While the position map takes asymptotically $O(N)$ space to store, in real-world deployments, the position map is typically small (e.g., less than 2.3 GB as shown in Table IV) and smaller than or comparable to the size of the eviction cache. For theoretic interest, it is possible to store the position map recursively in a smaller ORAM on the server, to reduce the client's local storage to sub-linear – although this is rarely necessary in practice.

B. Synchronous Amortized Shuffling Algorithm

The basic SSS construction as shown in Figure 2 employs for each partition an ORAM scheme (referred to as the partition ORAM) based on the original hierarchical construction by Goldreich and Ostrovsky [14], and geared towards optimal practical performance.

Such a partition ORAM requires periodic shuffling oper-

ations: every 2^i accesses to a partition ORAM, 2^i blocks need to be reshuffled for this partition ORAM. Reshuffling can take $O(\sqrt{N})$ time in the worst case, and all subsequent data access requests are blocked waiting for the reshuffling to complete.

Therefore, although the basic SSS construction has $O(\log N)$ amortized cost (non-recursive version), the worst-case cost of $O(\sqrt{N})$ makes it undesirable in practice. To address this issue, Stefanov *et. al.* propose a technique that spreads the shuffling work across multiple data accesses, to avoid the poor worst-case performance.

On a high level, the idea is for the client to maintain a shuffling job queue which keeps track of partitions that need to be reshuffled, and the respective levels that need to be reshuffled. A scheduler schedules $O(\log N)$ amount of shuffling work to be performed with every data access.

Stefanov *et. al.* devise a method for data accesses to nonetheless proceed while a partition is being shuffled, or pending to be reshuffled. Suppose that the client needs to read a block from a partition that is currently being shuffled or pending to be shuffled. There are two cases:

Case 1. The block has been fetched from the server earlier, and exists in one of the local data structures: the eviction cache, the shuffling buffer, or the storage cache. In this case, the client looks up this block locally. To prevent information leakage, the client still needs to read a fake block from every non-empty level in the server's partition. Specifically,

- For levels currently marked for shuffling, the client prefetches a previously unread block which needs to be read in for reshuffling (referred to as an *early cache-in*) – unless all blocks in that level have been cached in.
- For levels currently not marked for shuffling, the client requests a dummy block, referred to as a *dummy cache-in*.

Case 2. **The block has not been fetched earlier**, and resides in the server partition. In this case, the client reads the real block from the level where the block resides in, and for every other non-empty level, the client makes a fake read (i.e., early cache-in or dummy cache-in), using the same fake read algorithm described above.

IV. Formal Definitions

Traditional ORAMs assume synchronous I/O operations, i.e., I/O operations are blocking, and a data request needs to wait for a previous data request to end. To increase the amount of I/O parallelism, we propose to make I/O operations asynchrnous in ORAMs, namely, there can be multiple outstanding I/O requests, and completion of I/O requests are handled through callback functions.

Making ORAM operations asynchronous poses a security challenge. Traditional synchronous ORAM requires that the physical addresses accessed on the untrusted storage server must be independent of the data access sequence.

In asynchronous ORAM, the security requirement is complicated by the fact that the scheduling of operations is no longer sequential or blocking. There can be many ways to schedule these operations, resulting in variable sequences of server-observable events (e.g., I/O requests). Not only must the sequence of addresses accessed be independent of the data access sequence, so must the timing of these events.

We now formally define asynchronous (distributed) Oblivious RAM. For both the non-distributed and distributed case, we first define the set of all network or disk I/O events (including the timing of the events) observable by an adversary. The security definition or an asynchronous (distributed) ORAM intuitively says that *the set of events observable by the adversary should not allow the adversary to distinguish two different data request sequences of the same length and timing*.

Asynchronous ORAM. An asynchronous ORAM consists of a client, a server, and a network intermediary. Let seq denote a data access sequence:

$$\mathsf{seq} := [(\mathsf{blockid}_1, t_1), (\mathsf{blockid}_2, t_2), \ldots, (\mathsf{blockid}_m, t_m)]$$

where each $\mathsf{blockid}_i$ denotes a *logical* block identifier, and each t_i denotes the time of arrival for this request. Given any data access sequence seq, the ORAM client interacts with the server to fetch these blocks. Let

$$\mathsf{events} := [(\mathsf{addr}_1, \tau_1), (\mathsf{addr}_2, \tau_2), \ldots, (\mathsf{addr}_c, \tau_c)] \quad (1)$$

denote the event sequence resulting from a data access sequence, where each addr_i denotes a requested *physical* address on the server storage, and τ_i denotes the time at which the request is sent from the client.

We assume that the network and the storage are both under the control of the adversary, who can introduce arbitrary delays of its choice in packet transmissions and responses to requests.

Distributed asynchronous ORAM. A distributed asynchronous ORAM consists of multiple distributed trusted components which can communicate with each other, and communicate with untrusted storage servers. The adversary is in control of the storage servers, as well as all network communication. Although in practice, the storage servers are typically also distributed, for the security definitions below, we consider all untrusted storage servers as a unity – since they are all controlled by the adversary. In this section, we consider the abstract model of distributed asynchronous ORAM, while possible real-world instantiations are described in Section VI.

For a distributed asynchronous ORAM, We can define the sequence of all events to be composed of 1) all I/O requests (and their timings) between a trusted component to the untrusted storage; and 2) all (encrypted) messages (and

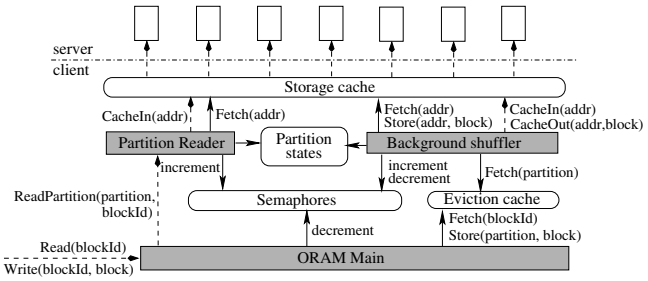


Figure 3: **Overview of asynchronous ORAM algorithm.** Solid arrows: synchronous calls. Dotted arrows: asynchronous calls.

their timings) between two trusted components:

$$\mathsf{events} := \left[\begin{array}{l} (\mathsf{addr}_1, \tau_1, \kappa_1), (\mathsf{addr}_2, \tau_2, \kappa_2), \ldots, (\mathsf{addr}_c, \tau_c, \kappa_c), \\ (\mathsf{m}_1, \tilde{\tau}_1, \kappa_1, \kappa_1'), (\mathsf{m}_2, \tilde{\tau}_2, \kappa_2, \kappa_2'), \ldots, (\mathsf{m}_d, \tilde{\tau}_d, \kappa_d, \kappa_d') \end{array} \right] \tag{2}$$

where $(\mathsf{addr}_i, \tau_i, \kappa_i)$ denotes that trusted component κ_i requests physical address addr_i from untrusted storage at time τ_i; and $(\mathsf{m}_i, \tilde{\tau}_i, \kappa_i, \kappa_i')$ denotes that trusted component κ_i sends an encrypted message m to trusted component κ_i' at time $\tilde{\tau}_i$.

Similarly to the non-distributed case, we say that a distributed asynchronous ORAM is secure, if an adversary (in control of the network and the storage) cannot distinguish any two access sequences of the same length and timing from the sequence of observable events.

Definition 1 (Oblivious accesses and scheduling)**.** *Let* seq_0 *and* seq_1 *denote two data access sequences of the same length and with the same timing:*

$$\mathsf{seq}_0 := \left[(\mathsf{blockid}_1, t_1), (\mathsf{blockid}_2, t_2), \ldots, (\mathsf{blockid}_m, t_m)\right],$$
$$\mathsf{seq}_1 := \left[(\mathsf{blockid}_1', t_1), (\mathsf{blockid}_2', t_2), \ldots, (\mathsf{blockid}_m', t_m)\right]$$

Define the following game with an adversary who is in control of the network and the storage server:

- *The client flips a random coin* b*.*
- *Now the client runs distributed asynchronous ORAM algorithm and plays access sequence* seq_b *with the adversary.*
- *The adversary observes the resulting event sequence and outputs a guess* b' *of* b*.*

We say that a (distributed) asynchronous ORAM is secure, if for any polynomial-time adversary, for any two sequences seq_0 *and* seq_1 *of the same length and timing,* $\left|\Pr[b' = b] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$*. where* λ *is a security parameter, and* negl *is a negligible function. Note that the set of events observed by the adversary in the non-distributed and distributed case are given in Equations 1 and 2 respectively.*

V. Asynchronous ORAM Construction

We now describe how to make the SSS ORAM asynchronous. This section focuses on the non-distributed case first. The distributed case is described in the next section.

Table II: Data structures used in ObliviStore

Data structure	Purpose
eviction cache	Temporarily caches real reads before eviction.
position map	Stores the address for each block, including which partition and level each block resides in.
storage cache	Temporarily stores blocks read in from server for shuffling, including early cache-ins and shuffling cache-ins. Also temporarily stores blocks after shuffling intended to be written back to the server.
shuffling buffer	Used for locally permuting data blocks for shuffling.
partition states	stores the state of each partition, including which levels are filled, information related to shuffling, and cryptographic keys.

A. Overview of Components and Interfaces

As shown in Figure 3, our basic asynchronous ORAM has three major functional components, the ORAM main algorithm, the partition reader, and the background shuffler.

ORAM main. ORAM main is the entry point to the ORAM algorithm, and takes in asynchronous calls of the form $\mathbf{Read}(\mathsf{blockid})$ and $\mathbf{Write}(\mathsf{blockid}, \mathsf{block})$. Response to these calls are passed through callback functions.

The ORAM main handler looks up the position map to determine which partition the requested block resides in, calls the partition reader to obtain the block asynchronously, and places the block in a freshly chosen random eviction cache. If the request is a write request, the block is overwritten with the new data before being placed in the eviction cache. The ORAM main handler then updates the position map accordingly.

Partition reader. The partition reader is chiefly in charge of reading a requested block from a chosen partition. It takes in asynchronous calls of the form $\mathbf{ReadPartition}(\mathsf{partition}, \mathsf{blockid})$, where responses are passed through callback functions.

Background shuffler. The background shuffler is in charge of scheduling and performing the shuffling jobs. Details of the background shuffler will be presented in Section V-E.

B. Data Structures and Data Flow

Table II summarizes the data structures in our ORAM construction, including the eviction cache, position map, storage cache, shuffling buffer, and partition states.

Informally, when a block is read from the server, it is first cached by the storage cache. Then, this block is either directly fetched into the shuffling buffer to be reshuffled; or it is passed along through the partition reader to the ORAM main handler, as the response to a data access request. The ORAM main handler then adds the block to an eviction cache, where the block will reside for a while before being fetched into the shuffling buffer to be reshuffled. Reshuffled blocks are then written back to the server asynchronously (unless they are requested again before being written back).

Table I: Types of cache-ins: when and for what purposes blocks are being read from the server.

Type of cache-in	Explanation
Early cache-in	**[Partition reader]** Early cache-in is when the client reads a block needed for shuffling over a normal data access. Specifically, when the partition reader tries to read a block from a partition that is currently being shuffled: if a level being shuffled does not contain the requested block, or contains the requested block but the requested block has already been cached in earlier, the client caches in a previously unread block that needs to be read for shuffling. The block read could be real or dummy.
Shuffling cache-in	**[Background shuffler]** Shuffling cache-in is when a block is read in during a shuffling job.
Dummy cache-in	**[Partition reader]** During a normal data access , if a level is currently not being shuffled, and the requested block does not reside in this level, read the next dummy block from a pseudo-random location in this level.
Real cache-in	**[Partition reader]** During a normal data access, if the intended block resides in a certain level, and this block has not been cached in earlier, read the real block.

Below we explain the storage cache in more detail. The partition states will be explained in more detail in Section V-C. The remaining data structures in Table II have appeared in the original SSS ORAM.

Storage cache. Blocks fetched from the server are temporarily stored in the storage cache, until they are written back to the server. The storage cache supports two asynchronous operations, i.e., $\mathbf{CacheIn}(\mathsf{addr})$ and $\mathbf{CacheOut}(\mathsf{addr})$. Upon a $\mathbf{CacheIn}$ request, the storage cache reads from the server a block from address addr, and temporarily stores this block till it is cached out. Upon a $\mathbf{CacheOut}$ request, the storage cache writes back to the server a block at address addr, and erases the block from the cache.

Blocks are re-encrypted before being written back to the storage server, such that the server cannot link blocks based on their contents. The client also attaches appropriate authentication information to each block so it can later verify its integrity, and prevent malicious tampering by the untrusted storage (see full version [39]).

Additionally, the storage cache also supports two synchronous operations, i.e., $\mathbf{Fetch}(\mathsf{addr})$ and $\mathbf{Store}(\mathsf{addr}, \mathsf{block})$, allowing the caller to synchronously fetch a block that already exists in the cache, or to synchronously store a block to the local cache.

There are 4 types of cache-ins, as described in Table I.

C. ORAM Partitions

Each partition is a smaller ORAM instance by itself. We employ a partition ORAM based on the hierarchical construction initially proposed by Goldreich and Ostrovsky [14], and specially geared towards optimal practical performance. Each partition consists of $\frac{1}{2}\log N + 1$ levels, where level i can store up to 2^i real blocks, and 2^i or more dummy blocks.

For each ORAM partition, the client maintains a set of partition states as described below.

Partition states. Each partition has the following states:

- A counter C_p. The value of $C_p \in [0, \mathsf{partition\_capacity})$ signifies the state of partition p. Specifically, let $C_p := \sum_i b_i \cdot 2^i$ denote the binary representation of the counter C_p corresponding to partition p. This means that the state of the partition p should be as below: 1) for every non-zero bit b_i, level i of the partition is filled on the server; and 2) for every bit $b_i = 0$, level i is empty.
- Job size J_p, which represents how many blocks (real or dummy) are scheduled to be written to this partition in the next shuffle. J_p is incremented every time a partition p is scheduled for an eviction. Notice that the actual eviction and the associated shuffling work may not take place immediately after being scheduled.
- A bit $\mathsf{bShuffle}$, indicating whether this partition is currently being shuffled.
- Dummy counters. Each partition also stores a dummy block counter for each level, allowing a client to read the next a previously unread dummy block (at a pseudo-random address).
- Read/unread flags. For every non-empty level, we store which blocks remain to be read for shuffling.

Batch shuffling. In the SSS ORAM algorithm [40], a new shuffling job is created whenever a block is being written to a partition – as shown in Figure 4 (left). The SSS algorithm performs these shuffling jobs sequentially, one after another. Notice that a new shuffling job can be created while the corresponding partition is still being shuffled. Therefore, the SSS algorithm relies on a shuffling job queue to keep track of the list of pending shuffling jobs.

As a practical optimization, we propose a method to batch multiple shuffling jobs together (Figure 4 – right). When a shuffling job is being started for a partition p, let C_p denote the current partition counter. Recall that the binary representation of C_p determines which levels are filled for partition p. Let J_p denote the current job size for partition p. This means that upon completion of this shuffling, the partition counter will be set to $C_p + J_p$. Furthermore, the binary representation of $C_p + J_p$ determines which levels are filled after the shuffling is completed. The values of C_p and J_p at the start of the shuffling job jointly determine which levels need to be read and shuffled, and which levels to be written to after the shuffling. Figure 4 (right) shows the idea behind batch shuffling.

New blocks can get scheduled to be evicted to partition p before its current shuffling is completed. ObliviStore does

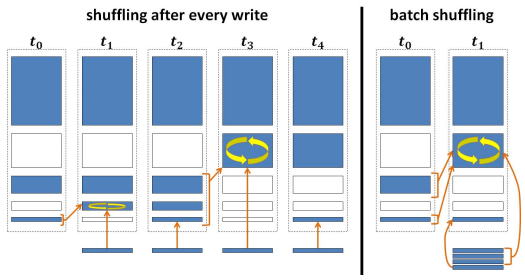


Figure 4: **Batch shuffling** – a new optimization technique for grouping multiple shufflings into one.

not try to cancel the current shuffling of partition p to accommodate the newly scheduled eviction. Instead, we continue to finish the current shuffling, and effectively queue the newly scheduled evictions for later shuffling. To do this, at the start of each shuffling, we *i)* take a snapshot of the job size: $\widehat{J}_p \leftarrow J_p$; and *ii)* set $J_p \leftarrow 0$. This way, we can still use J_p to keep track of how many new blocks are scheduled to be evicted to partition p, even before the current shuffling is completed.

D. Satisfying Scheduling Constraints with Semaphores

Our asynchronous ORAM construction must decide how to schedule various operations, including when to serve data access requests, how to schedule shufflings of partitions, and when to start shuffling jobs.

Constraints. We wish to satisfy the following constraints when scheduling various operations of the ORAM algorithm.

- *Client storage constraint.* The client's local storage should not exceed the maximum available amount. Particularly, there should not be too many early reads, shuffling reads, or real reads.
- *Latency constraint.* Data requests should be serviced within bounded time. If too many shuffling jobs are in progress, there may not be enough client local storage to serve a data access request, causing it to be delayed.

Semaphores. To satisfy the aforementioned scheduling constraints different components rely on semaphores to coordinate with each other. In our ORAM implementation, we use four different types of semaphores, where each type indicates the availability of a certain type of resource.

1) *early cache-ins semaphore*, indicating how many remaining early cache-ins are allowed,
2) *shuffling buffer semaphore*, indicating how many more blocks the shuffling buffer can store,
3) *eviction semaphore*, indicating how much data access is allowed to stay ahead of shuffling. This semaphore is decremented to reserve "evictions" as a resource before serving a data access request; and is incremented upon the eviction of a block (real or dummy) from the eviction cache.
4) *shuffling I/O semaphore*, indicating how much more I/O work the background shuffler is allowed to perform. This semaphore defines how much the shuffler is allowed to stay ahead of the normal data accesses, and prevents too much shuffling work from starving the data accesses.

Among the above semaphores, the early cache-in, shuffling buffer, and eviction semaphores are meant to bound the amount of client-side storage, thereby satisfying the *client storage constraint*. For early cache-ins and shuffling cache-ins, we bound them by directly setting a limit on the cache size, i.e., how many of them are allowed to be concurrently in the cache. The eviction semaphore mandates how much data accesses are allowed to stay ahead of shuffling – this in some sense is bounding the number of real blocks in the eviction cache. As explained later, due to security reasons, we cannot directly set an upper bound on the eviction cache size as in the early cache-in and shuffling buffer semaphores. Instead, we bound the number of real blocks indirectly by pacing the data accesses to not stay too much ahead of shuffling work. Finally, the shuffling I/O semaphore constrains how much shuffling I/O work can be performed before serving the next data access request. This is intended to bound the latency of data requests.

Preventing information leakage through semaphores. One challenge is how to prevent information leakage through semaphores. If not careful, the use of semaphores can potentially leak information. For example, when reading blocks from the server, some blocks read are dummy, and should not take space on the client-side to store. In this sense, it may seem that we need to decrement a semaphore only when a real block is read from the server. However, doing this can potentially leak information, since the value of the semaphore influences the sequence of events, which the server can observe.

Invariant 1 (Enforcing oblivious scheduling)**.** *To satisfy the oblivious scheduling requirement, we require that the values of semaphores must be independent of the data access sequence. To achieve this, operations on semaphores, including incrementing and decrementing, must depend only on information observable by an outside adversary who does not now the data request sequence.*

For example, this explains why the eviction semaphore does not directly bound the eviction cache size as the early cache-in and shuffling buffer semaphores do – since otherwise the storage server can potentially infer the current load of the eviction cache, thereby leaking sensitive information. To address this issue, we design the eviction semaphore not to directly bound the amount of eviction cache space available, but to pace data accesses not to stay too much ahead of shuffling. The SSS paper theoretically proves that if we pace the data accesses and shuffling appropriately, the eviction cache load is bounded by $O(\sqrt{N})$ with high probability.

E. Detailed Algorithms

The ORAM main, partition reader, and background shuffler algorithms are detailed in Figures 5, 6, and 7 re-

spectively. We highlighted the use of semaphores in bold. Notice that all semaphore operations rely only on publicly available information, but not on the data request sequence – both directly or indirectly. This is crucial for satisfying the oblivious scheduling requirement, and will also be crucial for the security proof in the full version [39].

F. Security Analysis: Oblivious Scheduling

We now formally show that both the physical addresses accessed and the sequence of events observed by the server are independent of the data access sequence.

Theorem 1. *Our asynchronous ORAM construction satisfies the security notion described in Definition 1.*

In the full version [39], we formally show that an adversary can perform a perfect simulation of the scheduler without knowledge of the data request sequence. Specifically, both the timing of I/O events and the physical addresses accessed in the simulation are indistinguishable from those in the real world.

VI. Distributed ORAM

One naive way to distribute an ORAM is to have a single trusted compute node with multiple storage partitions. However, in this case, the computation and bandwidth available at the trusted node can become a bottleneck as the ORAM scales up. We propose a distributed ORAM that distributes not only distributes storage, but also computation and bandwidth.

Our distributed ORAM consists of an *oblivious load balancer* and multiple *ORAM nodes*. The key idea is to apply the partitioning framework (Section III) twice. The partitioning framework was initially proposed to reduce the worst-case shuffling cost in ORAMs [35, 40], but we observe that we can leverage it to *securely* perform load balancing in a distributed ORAM. Specifically, each ORAM node is a "partition" to the oblivious load balancer, which relies on the partitioning framework to achieve load balancing amongst multiple ORAM nodes. Each ORAM node has several storage partitions, and relies on the partitioning framework again to store data blocks in a random storage partition with every data access. One benefit of the distributed architecture is that multiple ORAM nodes can perform shuffling in parallel.

A. Detailed Distributed ORAM Construction

To access a block, the oblivious load balancer first looks up its position map, and determines which ORAM node is responsible for this block. The load balancer than passes the request to this corresponding ORAM node. Each ORAM node implements a smaller ORAM consisting of multiple storage partitions. Upon obtaining the requested block, the ORAM node passes the result back to the oblivious load balancer. The oblivious load balancer now temporarily places the block in its eviction caches. With every data access, the oblivious load balancer chooses ν random ORAM nodes and evicts one block (possibly real or dummy) to each of them, through an ORAM write operation.

Each ORAM node also implements the shuffling functionalities as described in Section V. In particular, the ORAM nodes can be regarded as a parallel processors capable of performing reshuffling in parallel. The oblivious load balancer need not implement any shuffling functionalities, since it does not directly manage storage partitions. Hence, even though the load balancer is a central point, its functionality is very light-weight in comparison with ORAM nodes which are in charge of performing actual cryptographic and shuffling work.

Notice that each ORAM node may not be assigned an equal amount of storage capacity. In this case, the probability of accessing or evicting to an ORAM node is proportional to the amount of its storage capacity. For ease of explanation, we assume that each storage partition is of equal size, and that each ORAM node may have different number of partitions – although in reality, we can also support partitions of uneven sizes in a similar fashion.

Theorem 2. *Our distributed asynchronous ORAM construction satisfies the security notion described in Definition 1.*

Proof: (sketch.) Similar to that of Theorem 1. Both the oblivious load balancer and the ORAM node algorithms are perfectly simulatable by the adversary, without having to observe the physical addresses accessed. The detailed proof is in the full version [39]. ■

B. Dynamic Scaling Up

Adding compute nodes. When a new ORAM node processor is being added to the system (without additional storage), the new ORAM node processor registers itself with the load balancer. The load balancer now requests existing ORAM nodes to hand over some of their existing their partitions to be handled by the new processor. To do this, the ORAM nodes also need to hand over part of their local metadata to the new processor, including part of the position maps, eviction caches, and partition states. The load balancer also needs to update its local metadata accordingly to reflect the fact that the new processor is now handling the reassigned partitions.

Adding compute nodes and storage. The more difficult case is when both new processor and storage are being added to the system. One naive idea is for the ORAM system to immediately start using the new storage as one or more additional partitions, and allow evictions to go to the new partitions with some probability. However, doing so would result in information leakage. Particularly, when the client is reading the new partition for data, it is likely reading a block that has been recently accessed and evicted to this partition.

We propose a new algorithm for handling addition of new ORAM nodes, including processor and storage. When a new ORAM node joins, the oblivious load balancer and the

ORAM main loop:

- **Decrement the early cache-in semaphore by the number of levels.** *// Reserve space for early cache-ins.*
- **Decrement the eviction semaphore by eviction rate.** *// Evictions must be performed later to release the "eviction resource".*
- Fetch the next data access request for blockid, look up the position map to determine that blockid is assigned to partition p.
- Call $\mathbf{ReadPartition}(p, \mathsf{blockid})$.
- On callback:
  - Store the block to the eviction cache (overwrite block if this is an ORAM write request).
  - Let ν denote the eviction rate. Choose ν partitions at random for eviction, by incrementing their respective job sizes: $J_p \leftarrow J_p + 1$ *// If ν is a floating number, choose at least ν partitions on average.*

Figure 5: ORAM main algorithm.

$\mathbf{ReadPartition}(p^*, \mathsf{blockid})$:

1) Looks up the position map to determine the level ℓ^* where blockid resides. If the requested block is a dummy or blockid is not found in partition p^*, then set $\ell^* \leftarrow \perp$.
2) For each level in partition p^* that satisfies one of the following conditions: **increment early cache-in semaphore by** 1:
   - the level is empty;
   - the level is not marked for shuffling; or
   - the level is marked for shuffling but all blocks have been cached in.
3) For each filled level ℓ in partition p^*:
   - If $\ell = \ell^*$, $\mathbf{ReadReal}(\ell, \mathsf{blockid})$. Else, $\mathbf{ReadFake}(\ell)$.

$\mathbf{ReadReal}(\ell, \mathsf{blockid})$:

- If blockid has been cached in:
  - Call $\mathbf{ReadFake}(\ell)$
  - On completion of the fake (i.e., dummy or early) cache-in: return contents of blockid through callback. */* To prevent timing channel leakage, must wait for fake cache-in to complete before returning the block to ORAM main. */*
- Else:
  - Cache in block blockid from server.
  - On completion of cache-in, return contents of blockid through callback.

$\mathbf{ReadFake}(\ell)$:

- If level ℓ is not being shuffled:
  - Get address addr of next random dummy block.
  - Cache in the dummy block at addr.
- If level ℓ is being shuffled, and level ℓ has unread blocks,
  - Perform an early cache-in.
- Else return with $\perp$.

Figure 6: Partition reader algorithm.

new ORAM node jointly build up new storage partitions. At any point of time, only one storage partition is being built. Building up a new storage partition involves:

- *Random block migration phase*. The load balancer selects random blocks from existing partitions, and migrates them to the new partition. The new partition being built is first cached in the load balancer's local trusted memory, and it will be sequentially written out to disk when it is ready. This requires about $O(\sqrt{N/D})$ amount of local memory, where N is the total storage capacity, and D is the number of ORAM nodes.
  During the block migration phase, if a requested block resides within the new partition, the load balancer fetches the block locally, and issues a dummy read to a random existing partition (by contacting the corresponding ORAM node). Blocks are only evicted to existing partitions until the new partition is fully ready.
- *Marking partition as ready*. At some point, enough blocks would have been migrated to the new partition. Now the load balances sequentially writes the new partition out to disk, and marks this partition as ready.
- *Expanding the address space.* The above two steps migrate existing blocks to the newly introduced partition, but do not expand the capacity of the ORAM. We need to perform an extra step to expand ORAM's address space.
  Similarly, the challenge is how to do this securely. Suppose the old address space is $[1, N]$, and the new address space after adding a partition is $[1, N']$, where $N' > N$. One naive idea is to randomly add each block in the delta address space $[N+1, N']$ to a random partition. However, if the above is not an atomic operation, and added blocks become immediately accessible, this can create an information leakage. For example, after the first block from address space $[N+1, N']$ has been added, at this time, if a data access request wishes to fetch the block added, it would definitely visit the partition where the block was added. To address this issue, our algorithm first assigns each block from address space $[N+1, N']$ to a random partition – however, at this point, these blocks are not accessible yet. Once all blocks from address space $[N+1, N']$ have been assigned, the load balancer notifiers all ORAM nodes, and at this point, these additional blocks become fully accessible.

Initially, a new ORAM node will have 0 active partitions. Then, as new storage partitions get built, its number of active partitions gradually increases. Suppose that at some point

Background shuffler loop:

1) Start shuffling.
   - Find a partition p whose bShuffle indicator is 0 and job size $J_p > 0$. Start the shuffling of partition p.
   - Set bShuffle $\leftarrow 1$ for partition p. *// Each partition can only have one active shuffling job at a time.*
   - Mark levels for shuffling.
   - Take a snapshot of the partition job size $\widehat{J}_p \leftarrow J_p$.
2) Cache-in and reserve space.
   - For each unread block B in each level marked for shuffling:
     - **Decrement: 1) shuffling buffer semaphore, and 2) shuffling I/O semaphore;**
     - Issue a $\mathbf{CacheIn}$ request for B.
   - Let r denote the number of reserved slots in shuffling buffer so far. Let w denote the number of cache-outs that will be performed after this partition is shuffled. Note that $r \leq w$.
     Decrement the shuffling buffer semaphore by $w - r$.
     // Reserve space in shuffling buffer for early cache-ins, unevicted blocks, and dummy blocks.
3) Upon completion of all cache-ins, perform atomic shuffle.
   / Since computation is much cheaper than bandwidth or latency, we assume that the local shuffle is done atomically. */*
   - Fetch.
     - Fetch from the storage cache all cached-in blocks for levels marked for shuffling. For each cache-in fetched that is an early cache-in, **increment the early cache-in semaphore**.
       // These early cache-ins are now accounted for by the shuffling buffer semaphore.
     - Let $\widehat{J}_p$ denote the job size at the start of this shuffling. Fetch $\widehat{J}_p$ blocks from the eviction cache corresponding to the partition. **Increment eviction cache semaphore by $\widehat{J}_p$.**
       / If fewer than $\widehat{J}_p$ blocks for this partition exists in the eviction cache, the eviction cache returns dummy blocks to pad. These unevicted cache blocks are now accounted for by the shuffling buffer semaphore. */*
   - Shuffle.
     - Add dummies to the shuffling buffer to pad its size to w.
     - Permute the shuffling buffer.
   - Store.
     - Store shuffled blocks into storage cache: for each level ℓ, store exactly $2 \cdot 2^\ell$ blocks from the shuffling buffer (at least half of which are dummy). Mark destination levels as filled.
   - Unmark levels for shuffling. Set partition counter $C_p \leftarrow (C_p + \widehat{J}_p)$ mod partition_capacity. Clear bShuffle $\leftarrow 0$.
4) Cache-out.
   - For each block B to be cached out:
     - **Decrement the shuffling I/O semaphore.**
     - Issue a $\mathbf{CacheOut}$ call for block B.
   - On each cache-out completion: **increment the shuffling buffer semaphore**.

Figure 7: Background shuffler algorithm.

of time, each existing ORAM node has $c_1, c_2, \ldots, c_{m-1}$ partitions respectively, and the newly joined ORAM node has c_m active partitions, while one more partition is being built. Suppose all partitions are of equal capacity, then the probability of evicting to each active partition should be equal. In other words, the probability of evicting to the i'th ORAM node (where $i \in [m]$) is proportional to c_i.

The remaining question is when to stop the migration and mark the new partition as active. This can be done as follows. Before starting to build a new partition, the oblivious load balancer samples a random integer from the binomial distribution $k \xleftarrow{\$} B(N, \rho)$, where N is the total capacity of the ORAM, and $\rho = \frac{1}{P+1}$, where P denotes the total number of active partitions across all ORAM nodes. The goal is now to migrate k blocks to the new partition before marking it as active. However, during the block migration phase, blocks can be fetched from the new partition but not evicted back to it. These blocks fetched from the new partition during normal data accesses are discounted from the total number of blocks migrated.

The full node join algorithm in the full version [39].

VII. Experimental Results

We implemented ObliviStore in C#. The code base has a total of ~ 9000 lines of code measured with SLOCCount [3].

Eliminating effects of caching. We eliminate OS-level caching so that our experiments represent worst-case scenarios. Our implementation uses kernel APIs that directly access data on the physical disks and we explicitly disable OS-level caching for both disk reads and writes.

Warming up ORAMs. In all experiments, we warm up the ORAMs first before taking measurements. Warming up is achieved by always first initializing ObliviStore into a state that it would be after $O(N)$ accesses.

A. Single Client-Server Setting

1) Results with Rotational Hard Disk Drives: We ran experiments with a single ORAM node with an i7-930 2.8

Ghz CPU and 7 rotational WD1001FALS 1TB 7200 RPM HDDs with 12 ms random I/O latency [1]. To be comparable to PrivateFS, our experiments are performed over a network link simulated to have 50ms latency (by delaying requests and responses). We also choose the same block size, i.e., 4KB, as PrivateFS.

Throughput and response time. Figure 8 shows the throughput of our ORAM against the ORAM capacity. *For a 1TB ORAM, our throughput is about* 364*KB/s*. Figure 9 plots the *response time* for data requests with various ORAM capacities. *For a 1TB ORAM, our response time is about* 196*ms*. We stress that the response time is measured under maximum load – therefore, the response time accounts for both the online data retrieval and the offline shuffling overhead.

In both Figures 8 and 9, we also marked data points for PrivateFS and PD-ORAM for comparison. For a 1 TB ORAM, ObliviStore has about 18 times higher throughput than PrivateFS. Note that we set up this experiment and parameters to best replicate the exact setup used in the PrivateFS and PD-ORAM experiments [47].

Small number of seeks. Our optimizations for reducing disks seeks (see full version [39]) help greatly in achieving (relatively) high performance. Figure 16 plots the average number of seeks per ORAM operation. At 1TB to 10TB ORAM capacities, ObliviStore requires under 10 seeks per ORAM operation on average.

Effect of network latency. In Figures 10 and 14, we measure the throughput and latency of a 1 TB ObliviStore ORAM under different network latencies. The results suggest that for rotational hard drives, the throughput of ObliviStore is almost unaffected until about 1 second of network latency. To obtain higher throughput beyond 1s network latency, we can increase the level of parallelism in our implementation, i.e., allowing more concurrent I/Os – but this will lead to higher response time due to increased queuing and I/O contention.

The response time of ObliviStore (single node with 7 HDDS) is consistently 140ms to 200ms plus the round-trip network latency. The additional 140ms to 200ms is due to disk seeks, request queuing, and I/O contention.

2) Results with Solid State Drives: Even though our implementation makes a lot of progresses in reducing disk seeks, there are still about 4 to 10 seeks per ORAM operation on average (Figure 16). Solid state drives (SSDs) are known to perform much better with seek intensive workloads, but are also currently more expensive per GB of storage than HDDs. To compare HDD and SSD storage, we repeated the experiments of Section VII-A with 2 x 1TB solid state drives on Amazon EC2 using a hi1.4xlarge VM instance.

The results are shown in Figures 11, 12, and 13. In comparison, the throughput of ObliviStore with 2 SSDs of storage is about 6 to 8 times faster than with 7 HDD. For a typical 50ms network link, the response time with SSD storage is about half of that with HDD storage.

HDDs or SSDs? Our experiments suggest that roughly 21 to 28 HDDs can achieve the same throughput as a single SSD. Since the SSDs used in the experiment are about 20 times more expensive than the HDDs, for a fixed throughput, SSDs are slightly cheaper than HDDs. On the other hand, HDDs are about 20 times cheaper per unit of capacity. Under a typical 50ms network latency, SSDs halve the response time in comparison with HDDs.

B. Distributed Setting

We measure the scalability of ObliviStore in a distributed setting. We consider a deployment scenario with a distributed TCB in the cloud. We assume that the TCB is established through techniques such as Trusted Computing, and that the TCB is running on a modern processor. How to implement code attestation to establish such a distributed TCB has been addressed in orthogonal work [26, 27, 32, 33], and is not a focus of this evaluation.

For the distributed SSD experiments, each ORAM node was a hi1.4xlarge Amazon EC instance with 2x1TB SSDs of storage directly attached, and the load balancer ran on a cc1.4xlarge instance. Although our instances have 60GB of *provisioned* RAM, our implementation used far less (under 3 GB per ORAM node, and under 3.5 GB for the load balancer). The load balancer and the ORAM nodes communicate through EC2's internal network (under 5ms network latency).

Figure 15 suggests that the throughput of ObliviStore scales up linearly with the number of ORAM nodes, as long as we do not saturate the network. The total bandwidth overhead between the oblivious load balancer and all ORAM nodes is 2X, and we never saturated the network in all our experiments. For example, with 10 ORAM nodes and 4KB block size, the ORAM throughput is about 31.5 MB/s, and the total bandwidth between the load balancer and all ORAM nodes is about 63 MB/s. We also measured that ObliviStore's response time in the distributed setting is about 60ms for 4KB blocks and is mostly unaffected by the number of nodes (detailed results in the full version [39]).

The throughput of ObliviStore using HDD storage (also tested on Amazon EC2) similarly scales linearly with the number of nodes. Please refer to full version [39] for the concrete results.

C. I/O Bottleneck Analysis

I/O overhead. ObliviStore incurs about 40X-50X I/O overhead under parameters used in our experiments, i.e., to access one data block, on average 40-50 data blocks need to be accessed. Though this seems high, under the amount of ORAM capacity and private memory considered in this paper, the SSS scheme (what we implement) seems to achieve the lowest I/O overhead (absolute value instead of

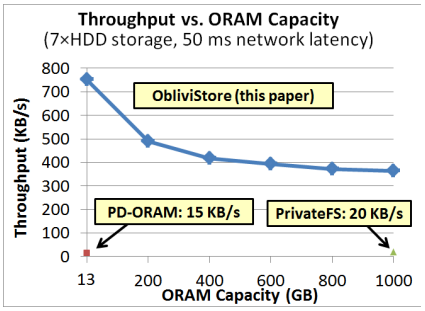


Figure 8: **ObliviStore throughput with 7 HDDs.** Experiment is performed on a single ORAM node with the following parameters: 50ms network latency between the ORAM node and the storage, 12ms average disk seek latency, and 4KB block size.

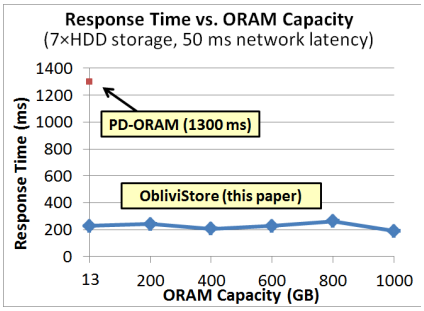


Figure 9: **ObliviStore response time with 7HDDs.** Experiment is performed on a single ORAM node with the following parameters: 50ms network latency between the ORAM node and the storage, 12ms average disk seek latency, and 4KB block size.

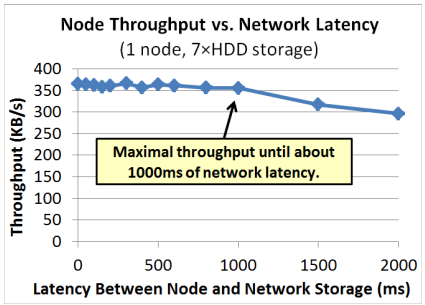


Figure 10: **Effect of network latency on throughput with 7HDDs.** Experiment is performed on a single ORAM node with 7 HDDs, 12ms average disk seek latency, and 4KB block size.

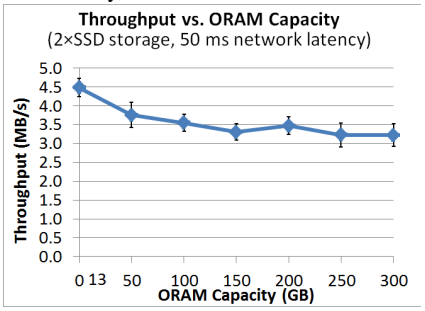


Figure 11: **ORAM throughput v.s. various ORAM capacities with 2SSDs.** The experiments are performed in a single client, single server setting with a simulated 50ms network link, and 2 SSDs attached to the server. Block size is 4KB.

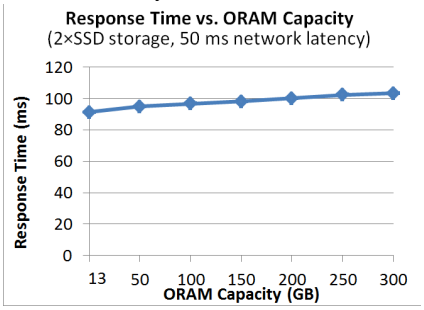


Figure 12: **ORAM response time v.s. various ORAM capacities with 2SSDs.** The experiments are performed in a single client, single server setting with a simulated 50ms network link, and 2 SSDs attached to the server. Block size is 4KB.

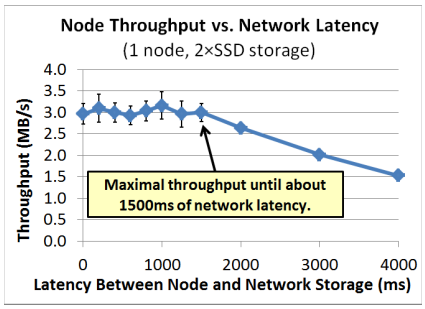


Figure 13: **Effect of network latency on throughput with 2 SSDs.** Experiment is performed on a single ORAM node with 2 SSDs and 4KB block size.

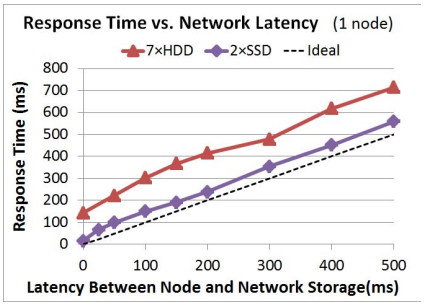


Figure 14: **Effect of network latency on response time.** Experiment is performed on a single ORAM node with 7 HDDs (12ms average seek latency), and again with 2 SSDs. Block size = 4KB. The ideal line represents the roundtrip network latency.

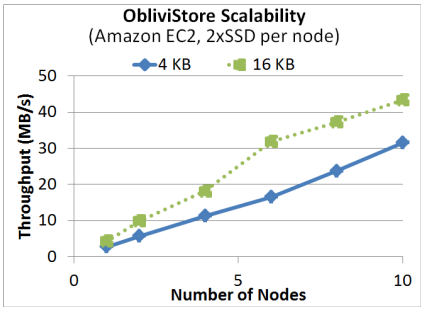


Figure 15: **Scalability of ObliviStore in a distributed setting.** 1 oblivious load balancer, 2 SDDs attached to each ORAM node. Throughput is the aggregate ORAM throughput at the load balancer which distributes the load across all ORAM nodes.

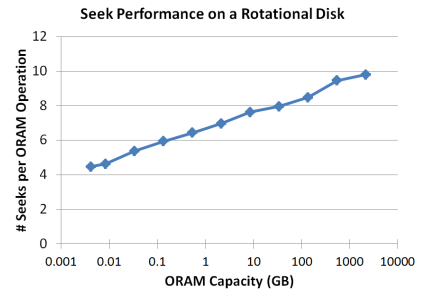


Figure 16: **Average number of seeks of ObliviStore per ORAM operation.** Includes all I/O to storage (reads and writes/shuffles). Experiment is performed on a single ORAM node with 4KB block size.

asymptotics) among all known ORAM schemes. Therefore, this is essentially the cost necessary to achieve the strong security of ORAM.

In comparison, PrivateFS should have higher I/O overhead – our I/O overhead is $O(\log N)$ with a constant under 2, while theirs is $O((\log N)(\log\log N)^2)$ [6]. This means that when network bandwidth is the bottleneck, PrivateFS achieves lower ORAM throughput than ObliviStore.

In our open source release, we will also implement the matrix compression optimization technique [40], which will further reduce the I/O overhead by a factor of 2.

Bottleneck analysis for various deployment scenarios. The I/O overhead means that for every 1MB/s ORAM throughput, we require about 40MB/s - 50MB/s throughput

on 1) AES computation, 2) total disk I/O bandwidth, and 3) total network bandwidth between ORAM nodes and disks.

Depending on the deployment scenario, one of the above three factors will hit bottleneck, which will become the main constraint on the ORAM throughput.

For the hybrid cloud setting, our experiments show that the network bandwidth between the private and public cloud is likely to be the bottleneck. For example, assuming a 1Gbps link between the private and public cloud, the network will become a bottleneck with a single ORAM node with 2 SSD drives – at the point of saturation, we would achieve roughly 25Mbps (or 3MB/s) ORAM throughput.

For the trusted hardware in the cloud scenario, assume that SSD drives are directly attached to ORAM nodes, and that the distributed TCB is running on modern processors (e.g., using Trusted Computing to establish a distributed TCB in the cloud)[1]. In this case, the bottleneck is likely to be disk I/O, since the total amount of data transferred between the oblivious load balancer and the ORAM nodes is relatively small, whereas the provisioned network bandwidth between them is large. Specifically, under our setup where each ORAM node has 2SSDs directly attached, suppose the network bandwidth is Zbps shared amongst the oblivious load balancer and all ORAM nodes, we can support roughly $20Z$ ORAM nodes before the network starts to be saturated. The total ORAM throughput should be $3.2y$MB/s, where $y < 20Z$ is the total number of ORAM nodes.

Our experiments suggest that computation is not the bottleneck when ORAM client algorithms (including the oblivious load balancer and the ORAM node algorithms) are run on a modern processor.

D. Applications

Oblivious file system. Using NBD (short for Network Block Device), we mounted the EXT4 File System on top of our ORAM (a single host with a single SSD). On top of this oblivious file system, we achieved average read/write throughput of roughly 4MB/s. For metadata operations, it took $2.1 - 3.5$ seconds to to create and delete 10,000 files. How to hide the number of accesses (e.g., depth of directory) is our future work.

E. Comparison with Related Work

The most comparable work is PrivateFS (PD-ORAM) by Williams *et. al.* [47]. Other than ObliviStore, PrivateFS is the most efficient ORAM implementation known-to-date. PrivateFS also propose a novel algorithm for multiple clients to share the same ORAM, while communicating amongst each other using a log on the server side.

Lorch *et. al.* also implement ORAM in a distributed data center setting [25]. They are the first to actually implement ORAM on off-the-shelf secure co-processors such as SLE 88 and IBM 4768, and therefore can achieve physical security which off-the-shelf trusted computing technologies (e.g, Intel TXT and AMD SVM) do not provide. On the other hand, their implementation is constrained by the chip I/O, computational power, and memory available in these secure co-processors. Lorch *et. al.* performed small-scale experiments with a handful of co-processors, and projected the performance of their distributed ORAM through theoretic calculations. Their work suggests that for ORAM to become practical in large-scale data centers, we need more powerful processors as part of the TCB. One way is to rely on Trusted Computing – although this does not offer physical security, it reduces attack surface by minimizing TCB such that formal verification may be possible. It is also conceivable that more powerful secure co-processors will be manufactured in the future [12]. Iliev and Smith also implemented an ORAM algorithm to create a tiny TCB [19] with secure hardware.

Table IV compares our work against related works. As mentioned earlier, since the work by Shroud [25] is less comparable, below we focus on comparing with PrivateFS [47]. The table suggests that on a single node with 7HDDs and under the various parameters used in the experiments, 1) ObliviStore achieves an order of magnitude higher throughput than PrivateFS; and 2) ObliviStore lowers the response time by 5X or more. Although we do not have access to their implementation, we conjecture that the speedup is partly due to the reduced number of disk seeks in our implementation (Figure 16, Section VII). Disk seeks are the main bottleneck with HDDs as the storage medium, since ORAM introduces a considerable amount of random disk accesses. While both schemes have $O(\log N)$ seeks in theory [6], ObliviStore is specifically optimized to reduce the number of seeks in practice. It is also likely that our implementation benefits from a finer granularity of parallelism, since we rely on asynchronous I/O calls and build our own optimized event scheduler. In comparison, PrivateFS uses multiple synchronous threads to achieve parallelism. Below are some additional remarks about the comparison between ObliviStore and PrivateFS:

- For ObliviStore, all HDD experiments consume under 30 MB/s (i.e., 240Mbps) network bandwidth (in many cases much less) – hence we never saturate a 1Gbps network link.
- For our HDD experiments, we had several personal communications [6] with the authors of PrivateFS to best replicate their experimental setup. Our disks have similar performance benchmarking numbers as theirs (approximately 12ms average seek time). We have also chosen our network latency to be 50ms to replicate their network characteristics. Both PrivateFS (PD-ORAM) and ObliviStore run on similar modern CPUs. *Our experiments show that CPU is not the bottleneck – but disk I/O is. The minor difference in the CPU is not crucial to the*

[1]For off-the-shelf secure co-processors such as IBM 4768, chip I/O and computation will be the main bottlenecks, as demonstrated by Lorch *et. al.* [25]. See Section VII-E for more details).

Scheme	Processors	Deployment scenario	Methodology	Bottleneck
Shroud [25]	secure co-processors	Trusted hardware in cloud	experiments and theoretic projection	chip I/O, computation power, and memory of secure co-processors
PrivateFS (PD-ORAM) [47]	modern CPUs	hybrid cloud	experiments	Disk I/O or Network I/O
ObliviStore	modern CPUs	both	experiments	Disk I/O or Network I/O

Table III: **Comparison of experimental setup.**

Scheme	Experimental setup			Results		
	block size	ORAM capacity	processors	private RAM consumed	response time	throughput
Secure co-processors (IBM 4764), distributed setting						
Shroud [25]	10 KB	320 TB	10,000*	300 GB	360 ms	28 KB/s
7 HDDs, 50ms network latency to storage, 12ms disk seek latency, single modern processor (client-side)						
PrivateFS‡ [6, 47]	4 KB	100MB	1	$<$ 2 GB †	$>$1s†	110 KB/s† (peak performance [6])
PD-ORAM‡ [47]	10 KB	13 GB	1	$<$ 2 GB†	$>$1s	15 KB/s
ObliviStore	4 KB			**0.46 GB**	**191 ms**	**757 KB/s**
PrivateFS‡ [6, 47]	4 KB	1 TB	1	$<$ 2 GB†	$>$1s	20 KB/s†
ObliviStore	4 KB			**2.3 GB**	**196 ms**	**364 KB/s**
Distributed setting, 20 SSDs, 11 modern processors 1 oblivious load balancer + 10 ORAM nodes (each with 2SSDs directly attached)						
ObliviStore	4 KB	3 TB	11	**36 GB**	**66 ms**	**31.5 MB/s**
ObliviStore	16 KB	3 TB	11	**33 GB**	**276 ms**	**43.4 MB/s**

Table IV: **Comparison with related work.**

Throughput means average total throughput measured after warming up the ORAM (i.e., the ORAM is in a state that it would be after $O(N)$ accesses, where N is the ORAM capacity), unless otherwise indicated.

†: These numbers obtained through personal communication [6] with the authors of PrivateFS [47]. PrivateFS reports the amount of private memory provisioned (instead of consumed) to be 2GB.

‡: Based on personal communication with the authors, the PrivateFS paper has two sets of experiments: PD-ORAM experiments and PrivateFS experiments. Based on our understanding: *i)* PD-ORAM seems to be an older version of PrivateFS; and *ii)* the experimental methodology for these two sets of experiments are different.

*: Based on a combination of experimentation and theoretic projection. Due to the constrained I/O bandwidth and computational power of IBM 4768 secure co-processors, unlike PrivateFS and ObliviStore, Shroud [25] is mainly constrained by the chip I/O, computational power, and memory available on these off-the-shelf secure co-processors.

performance numbers for ObliviStore.

- PrivateFS also experimented with faster disks, i.e., six 0.4TB 15K RPM SCSI (hardware RAID0) disks. They report a 2X speedup with these faster HDDs due to the superior seek time on these drives. We were not able to obtain the same disks for our experiments, but since disk seek is our main bottleneck with the HDD experiments, we expect to see a similar speedup with these faster disks.

VIII. Related Work

Oblivious RAM: theory. Oblivious RAM was first proposed by Goldreich and Ostrovsky [14]. They propose a seminal hierarchical construction with $O((\log N)^3)$ amortized cost, where N denotes the storage capacity of the ORAM. This means that to access a block, a client needs to access $O((\log N)^3)$ blocks on average to mask from the server the true block of intent. Since then, a line of research has been dedicated to ORAM [9, 11, 13–16, 18, 23, 28, 29, 31, 43, 44, 46], most of which build on top of and improve the original hierarchical construction by Goldreich and Ostrovsky [14]. Recently, researchers have proposed a new paradigm for constructing ORAM [35, 40]. By relying on secure partitioning, this new paradigm breaks an ORAM into smaller instances, therefore reducing data shuffling (i.e., oblivious sorting) overhead [40] or completely eliminating oblivious sorting [35]. Constant round-trip ORAMs have been studied in seminal works by Goodrich *et. al.* [17] and Williams *et. al.* [45].

Oblivious RAM: bridging theory and practice. Willams, Sion *et. al.* have been pioneers in bridging the theory and practice of ORAM [43, 46, 47]. Goodrich, Mitzenmacher, Ohrimenko, Tamassia *et. al.* [17, 18] have also made significant contributions to bridging the theory and practice of ORAM.

Backes *et. al.* [8] use a combination of the binary-tree ORAM [35] and trusted hardware to build privacy-preserving behavioral advertising applications. They demonstrated a request latency of 4 to 5 seconds under rea-

sonable parametrization. However, their implementation is synchronous and all operations are blocking and sequentialized. Backes *et. al.* reported only latency results, but no throughput results. Their request latency can be broken down into an online latency of 750ms for fetching data from ORAM, and an offline latency of ~ 4s for data shuffling.

The most closely related works are the independent works by Williams *et. al.* [47] (i.e., PrivateFS and PD-ORAM) and Lorch *et. al.* [25]. We refer the readers to Section VII-E for a detailed comparison of these works and ours.

ACKNOWLEDGMENTS

We gratefully acknowledge Dawn Song and Bobby Bhattacharjee for their kind support, Dustin Schnaitman from Amazon for helping us acquire resources, and Jonathan Dautrich for helping clarify the pseudocode. We are indebted to Radu Sion, Peter Williams, Jay Lorch, and Bryan Parno for patiently discussing the details of PrivateFS/Shroud with us, so we can make an informed comparison. We would also like to thank the anonymous reviewers for their insightful comments and suggestions. This material is partially supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0946797, and by the DoD National Defense Science and Engineering Graduate Fellowship. 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.

REFERENCES

[1] http://www.storagereview.com/php/benchmark/suite_v4.php?typeID=10&testbedID=4&osID=6&raidconfigID=1&numDrives=1&devID_0=368&devCnt=1.
[2] IBM 4764 PCI-X cryptographic coprocessor (PCIXCC). http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml.
[3] Sloccount. http://www.dwheeler.com/sloccount/.
[4] Ssd adoption in the real world. http://esj.com/blogs/enterprise-insights/2012/08/ssd-adoption.aspx.
[5] Trusted computing group. http://www.trustedcomputinggroup.org/.
[6] Personal communication with Radu Sion and Peter Williams., Nov. 2012.
[7] D. Asonov and J.-C. Freytag. Almost optimal private information retrieval. In *PET*, 2003.
[8] M. Backes, A. Kate, M. Maffe, and K. Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *S & P*, 2012.
[9] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.
[10] 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.
[11] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.
[12] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.
[13] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.
[14] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.
[15] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.
[16] 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.
[17] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *CODASPY*, 2012.
[18] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.
[19] A. Iliev and S. Smith. Towards tiny trusted third parties. Technical report, 2005.
[20] 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.
[21] 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.
[22] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *SOSP*, 2009.
[23] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.
[24] J. Liedtke. On micro-kernel construction. In *SOSP*, 1995.
[25] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. *FAST*, 2013:199–213, 2013.
[26] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. Trustvisor: Efficient TCB reduction and attestation. In *S & P*, 2010.
[27] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.
[28] R. Ostrovsky. Efficient computation on oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1990.
[29] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.
[30] D. Perry. SSD prices falling faster than HDD prices. http://www.tomshardware.com/news/ssd-hdd-solid-state-drive-hard-disk-drive-prices,14336.html, 2011.
[31] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.
[32] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*, 2004.
[33] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: a new abstraction for building trusted cloud services. In *Usenix Security*, 2012.
[34] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *CCSW*, pages 43–46, 2010.
[35] 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.
[36] S. W. Smith. Outbound authentication for programmable secure coprocessors. In *ESORICS*, 2002.
[37] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Syst. J.*, 40(3):683–695, Mar. 2001.
[38] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Comput. Netw.*, 31(9):831–860, 1999.
[39] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. Technical report.
[40] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.
[41] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels. Iris: A scalable cloud file system with efficient integrity checks. In *ACSAC*, 2012.
[42] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *EuroSys*, 2010.
[43] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.
[44] P. Williams and R. Sion. Round-optimal access privacy on outsourced storage. In *CCS*, 2012.
[45] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.
[46] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.
[47] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.
[48] K. Zhang, X. Zhou, Y. Chen, X. Wang, and Y. Ruan. Sedic: privacy-aware data intensive computing on hybrid clouds. In *CCS*, 2011.