1 Introduction

Cloud computing and cloud storage are becoming an attractive option for businesses and governmental organizations in need of scalable and reliable infrastructures. Cloud providers, e.g., Amazon or Google, have substantial expertise and resources, allowing them to rent their services at very competitive prices. Cloud users are drawn by the ability to pay for only what they need, but maintain the ability to scale up if requirements change. Users can now take advantage of highly reliable storage solutions without investing large amounts of money for data centers upfront.

Unfortunately, there is a significant downside to storing data in the cloud. Cloud providers cannot always be fully trusted and may not treat sensitive user data very carefully. Seeing news of high-profile hacking incidents involving data theft has become commonplace \cite{8,18}. Encryption of data at rest provides a partial solution to this problem, but it is not sufficient. Even if the cloud cannot read the encrypted data, it may be able to learn valuable information based on when and how often users access their data. We call this information the user’s “access pattern”. As a motivating example, consider a hospital that outsources patient records to the cloud in order to save on replication and IT costs. If the cloud server sees that, e.g., an oncologist accesses a patient’s data, they can learn with some degree of certainty that this patient has cancer. An adversary could slowly aggregate information on data accesses to learn potentially important secrets. As it is generally difficult to quantify what external knowledge adversaries may have and what inferences they could make, it is important to hide a user’s access pattern as well as the data being accessed.

There are traditionally two ways to hide a user’s access pattern: Oblivious RAM (ORAM) \cite{7} and Private Information Retrieval (PIR) \cite{9}. The traditional approach taken by ORAM is to arrange the data in such a way that the user never touches the same piece twice, without an intermediate “shuffle” which erases the correlation between block locations. ORAMs have historically featured low amortized communication complexity and did not require any computation on the server, but occasionally the user was required to download and reshuffle the entire database. This could become impractical in cloud scenarios, especially if the user is a low-powered or communication-constrained device.

Private Information Retrieval, in contrast with ORAM, hides the target of each individual query, independent of all previous queries. This can be accomplished, for example, by using a homomorphic encryption which the server uses to operate over the entire database, selecting out the block of data that the user has requested. The user generates encrypted requests and sends them to the server. Since PIR does not try to hide a sequence of accesses, but each access individually, the
amortized cost is equal to the worst-case cost. Unfortunately, the requirement that the server computes over the entire database for each query is often impractical, especially for large databases.

Fortunately, there has been a recent flurry of research on Oblivious RAM, achieving many improvements that were long thought difficult or impossible. Chief among them being sublinear worst-case complexity guarantees, accomplished by Shi et al. [13] and independently by Kushilevitz et al. [10], but with very different approaches. Since then, several additional schemes have been proposed that achieve better communication complexity, but at the cost of increasing client memory from constant to logarithmic [17] or polynomial [15, 16] in \( n \), the number of blocks in the database.

The works of Shi et al. [13] and Stefanov et al. [17] are especially interesting because they use an entirely new paradigm for Oblivious RAM: a tree-based construction in which data blocks are inserted at the root and incrementally filtered down to the leaf nodes, making room for future operations at the top of the tree. This has the benefit of giving their scheme a worst-case complexity that is equal to the average case, which is a significant gain in terms of practicality. It also allows for large gains in performance. However, the current leading ORAM construction, Path ORAM, might still be considered too costly to use in a real-world situation. For example, on common databases it may impose up to a 200x overhead on communication. Since cloud deployment is currently used to save money on infrastructure costs, any potential user must be very wary of bandwidth and computational overhead which could easily outweigh any cost savings.

In this thesis, we develop more practical solutions for privacy-preserving data outsourcing, which include lower overhead and improved costs when executed on cloud platforms. Beyond simple query overhead, there are also other important matters to consider before a system is ready for cloud deployment. Common cloud architectures are massively parallel and distributed, meaning that a successful protocol should be easily parallelizable and adaptable to distributed computing platforms, e.g. MapReduce.

Moreover, the change from amortized to worst-case complexity has also added an additional problem that impedes its widespread use: it is no longer easy to resize ORAMs. Previously, one could simply choose a new size when the database was being shuffled. Now, however, it would ruin the worst case complexity of the schemes to have to process the entire database at once.

## 2 Practical Private Information Retrieval on MapReduce

Our first contribution is an efficient adaptation of PIR for existing cloud platforms using MapReduce. In previous work, the server’s computation is comprised of expensive cryptographic operations over the entire database. Because of the significant overhead this imposes, it has been questioned whether PIR will ever become practical in a real-world cloud computing setting, cf. Sion and Carbunar [14]. Typically, cloud providers such as Amazon charge their customers for both data transfer and CPU hours [1]. Due to the necessary condition that PIR protocols compute over the entire database for each query, it has been argued that trivial PIR (retrieving the whole database of \( n \ell \)-bit elements) is not only faster, but also cheaper for the cloud customer compared to a PIR query that involves lengthy computation [3, 12, 14].

Another open question is how to perform PIR in a real-world cloud computing environment. In contrast to a single machine server storing the data, one of the biggest challenges in cloud computing is a design that scales easily to the large distributed systems which are characteristic in
cloud settings. In order to alleviate this difficulty, major cloud providers (e.g., Amazon, Google, IBM, Microsoft) offer an interface to the prominent MapReduce [6] API for distributed computing to their users. MapReduce comprises not only parallelization (“Map”) of work, but an aggregation (“Reduce”) of individual results to keep computational burden on the user side low.

From a security perspective, we consider the single-server, computationally-private information retrieval setting. This is appropriate, because, although a cloud provider may allow access to many servers, they must all be considered as one “trust entity”: they are under the control of the same organization. Trust-wise, the cloud as a single, large server with many distributed CPUs. In the single-server setting, it is known that unconditionally secure PIR cannot be achieved more efficiently than transferring the entire database [4], so we are concerned instead with a computationally secure protocol (cPIR). Further use of the term PIR will be in reference to computationally-secure PIR, unless otherwise noted.

We propose PIRMAP, an efficient single-server cPIR protocol suited for MapReduce clouds. PIRMAP especially targets retrieval of relatively large files, a more specific setting than considered by previous work. In a scenario with \( n \) files each of size \( \ell \) bits and \( l \gg n \), PIRMAP achieves communication complexity linear in \( \ell \) with low constants, and improved performance compared to related work. PIRMAP is also designed for and leverages MapReduce parallelization and aggregation.

Our strategy to privately retrieve one block out of an \( n \)-block database is relatively simple: using an additively homomorphic encryption scheme, \((E, D)\), the client generates a vector of \( n \) ciphertexts where one of them, corresponding to the index of the block they wish to retrieve, is equal to \( E(1) \) and all the rest are \( E(0) \). Since our encryption scheme is additively homomorphic, we have that

\[
E(a) \circ_1 E(b) = E(a + b)
\]

for some operator \( \circ_1 \). Subsequently, we also have that

\[
E(a) \circ_2 b = E(a \cdot b)
\]

where \( \circ_2 \) is repeated application of \( \circ_1 \). For instance, if we used Paillier encryption then \( \circ_1 \) is integer multiplication and \( \circ_2 \) is exponentiation. Therewith, the server can compute the “dot product” of the client’s vector with its database (replacing the usual multiplication with \( \circ_2 \) and the addition with \( \circ_1 \)) and obtain an encryption of the block which the client is interested in retrieving. All the blocks they are not interested in will be “zeroed out” by the \( E(0) \) elements in the request vector and the single block, which was multiplied by \( E(1) \), will remain. Summing all these ciphertexts will result in a single encryption of the requested block, which is returned to the client.

This technique of using additively homomorphic encryption is typical in PIR (in fact our scheme can be viewed as a variation on the original by Kushilevitz and Ostrovsky [9]), but sending one ciphertext per element in the database seems, on the surface, like more overhead than existing schemes. However, if the individual blocks in the database are large, the size of these request vectors can be comparatively quite small. For instance, if the blocks are of size 1 MB, and each ciphertext is 1024 bits, the database can be over 8 GB before the request vector becomes larger than a single block. Having one ciphertext per block also allows us to have maximum parallelization and, as seen in Table 1, can even save on communication costs in some cases due to fact that other schemes, e.g. Lipmaa [11], cause ciphertext expansion in the result returned to the client.
Table 1: Real communication costs of related work in MB for different database sizes.

<table>
<thead>
<tr>
<th></th>
<th>1 MB Files</th>
<th></th>
<th>5 MB Files</th>
<th></th>
<th>10 MB Files</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 GB</td>
<td>10 GB</td>
<td>100 GB</td>
<td>1 GB</td>
<td>10 GB</td>
<td>100 GB</td>
</tr>
<tr>
<td>Lipmaa</td>
<td>10</td>
<td>13</td>
<td>17</td>
<td>38</td>
<td>55</td>
<td>71</td>
</tr>
<tr>
<td>Aguilar-Gaborit</td>
<td>29</td>
<td>240</td>
<td>2350</td>
<td>35</td>
<td>77</td>
<td>499</td>
</tr>
<tr>
<td>PIRMAP</td>
<td>3</td>
<td>6</td>
<td>38</td>
<td>11</td>
<td>12</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 2: Query time in minutes, including generation, evaluation, and decryption, for databases of varying sizes, composed of 5 MB files.

<table>
<thead>
<tr>
<th></th>
<th>5 GB</th>
<th>10 GB</th>
<th>15 GB</th>
<th>20 GB</th>
<th>25 GB</th>
<th>30 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lipmaa</td>
<td>1852</td>
<td>3704</td>
<td>5508</td>
<td>7312</td>
<td>9116</td>
<td>10920</td>
</tr>
<tr>
<td>Aguilar-Gaborit</td>
<td>4</td>
<td>7</td>
<td>11</td>
<td>14</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>PIRMAP</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since we can arrange our computation as a dot product, it easily converts to a MapReduce algorithm. MapReduce consists of two phases, Map and Reduce. The first phase runs the Map algorithm over every piece of the input in parallel. In our case, this would be the multiplication part of the dot product: using $\odot_2$ on a request ciphertext and a piece of the database. Once that is done, Reduce combines the output of the map algorithm. For us, this consists of using $\odot_1$ to aggregate the output of the Map phase, resulting in a single encryption which is returned.

One advantage over related work, and what allows us to achieve good computational performance, is that we use modern encryption which has more efficient operators for $\odot_1$ and $\odot_2$. In our above example using Paillier, exponentiation can be very expensive. Instead, we allow $\odot_2$ to be multiplication and $\odot_1$ to be simple addition. This is made possible by the recent work in *somewhat homomorphic* encryption, which aims to allow both addition and multiplication of ciphertexts. A side effect of providing both additive and multiplicative homomorphisms is that these schemes have much more efficient additive homomorphisms which, as shown in Table 2, make our PIR orders of magnitude faster. We note that our linear scheme is well suited to somewhat homomorphic encryption schemes because it requires only one scalar multiplication. One downside of such schemes is that further multiplications can become quite expensive, making their application to Lipmaa’s protocol much more difficult and expensive.

Finally, we are able to experimentally show on Amazon Elastic MapReduce that both the execution time and cost of our PIR are substantially lower than the naive solution of transferring the entire database, in spite of predictions by Chen and Sion [3].

3 Combining Oblivious RAM and PIR

We have shown that, in practice, one can privately retrieve a block from a database faster and cheaper than simply downloading the entire database. Our above solution even has optimal bandwidth overhead when $\ell \gg n$. The downside is that it requires the server to compute over the entire database for each retrieval.

In light of these results, we propose Path-PIR, a new construction combining ORAM and PIR, thereby overcoming the individual drawbacks of each of the two approaches. Path-PIR’s strategy
is to replace costly bucket accesses in the recently proposed ORAM by Shi et al. [13] with more efficient PIR operations. As noted above, PIR can be efficient when the block size is large in relation to the number of elements in the database. Since the tree-based ORAM of Shi et al. [13] is composed of many buckets, each of which has only a small number of elements, we are able to effectively leverage the optimal communication properties of PIR, while at the same time using the tree structure to limit the portion of the database which is subject to expensive homomorphic operations.

Additionally, we explore the notion of an ORAM’s latency, the amount of communication required before the client has access to their data. This is important, because many modern ORAM constructions involve an initial query, which returns the data, and a more expensive “book keeping” protocol which performs shuffling of the data to ensure the integrity and obliviousness of the data structure. Low latency is a highly desirable feature, since the reorganizing and shuffling of the data structure can happen in the background after the client receives their data. We also note that low latency can be very useful in certain client settings with restrictive data limits, i.e., smart phones.

The ORAM scheme we base our work on [13] stores blocks in a binary tree structure with \( n \) leaf nodes. Every node in the tree is a “bucket”, which holds up to \( Z = \Theta(\log n) \) elements. These buckets are in turn smaller ORAMs, but their size is so small such that the naive “retrieve all and reencrypt” method of oblivious access more practical than any more advanced technique. The tree construction works by inserting blocks into the root node, tagged with a uniformly chosen identifier from 1 to \( n \), and slowly percolating them down the tree toward the leaf node specified by that identifier. Whenever a block is read, the path from root to leaf which is known to contain that block is read in its entirety, and the block is removed from the ORAM. It is then retagged with a new random identifier and inserted into the top of the tree. After every operation, there is an “eviction” phase where the client obliviously moves blocks down the tree, ensuring that there is always space at the top for new blocks to be added.

Our key observation is that the bucket ORAMs are retrieved using the bandwidth intensive “naïve” method, which we have already shown to be substantially less efficient than PIR. By replacing this retrieval with a PIR query, we go from \( O(\log n) \) bandwidth overhead to retrieve an element from a bucket ORAM to \( O(1) \), under good parameters. Additionally, since the number
Table 3: Communication complexity of Path-PIR and related constant-memory schemes. Here, $N$ is the ORAM capacity, e.g., the number of files, $l$ is the bit-length of each file, and $k$ is the security parameter. Latency is the amount of communication before the client has access to data. The “practical” setting is $\ell > 100\text{kb}$ and $N < 2^{35}$.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Latency</th>
<th>Worst-Case</th>
<th>Practical Worst-Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shi et al. [13]</td>
<td>$O(\ell \cdot \log^2(N))$</td>
<td>$O(\ell \cdot \log^2(N))$</td>
<td>$O(\ell \cdot \log^2(N))$</td>
</tr>
<tr>
<td>Kushilevitz et al. [10]</td>
<td>$O(\ell \cdot \log^2(N) \log(N))$</td>
<td>$O(\ell \cdot \log^2(N) \log(N))$</td>
<td>$O(\ell \cdot \log^3(N))$</td>
</tr>
<tr>
<td>Path-PIR Linear</td>
<td>$O(k \cdot \log(N) + \ell)$</td>
<td>$O(k \cdot \log^3(N) + \ell \cdot \log(N))$</td>
<td>$O(\ell \cdot \log(N))$</td>
</tr>
<tr>
<td>Path-PIR FHE</td>
<td>$O(k + l)$</td>
<td>$O(k \cdot \log(N) + l \cdot \log(N))$</td>
<td>$O(k + l)$</td>
</tr>
<tr>
<td>Optimal</td>
<td>$O(\log(N) + l)$</td>
<td>$O(\log(N) + l)$</td>
<td>$O(\log(N) + l)$</td>
</tr>
</tbody>
</table>

of entries in a bucket is very small, the range of “good” parameters is much less restrictive than in PIRMAP. In practice, as long as $\ell$ is greater than the size of a homomorphic ciphertext, we are saving bandwidth.

However, in addition to reading using PIR, we also have to be able to update the contents of the buckets. Fortunately, we can take advantage of a primitive called PIR-writing which provides the same security guarantees as PIR but in reverse: the client can update one block of the database without the server learning which block was updated. Our PIR scheme from before easily adapts into a PIR-writing scheme with the same asymptotic complexity, and we have a complete bucket construction which uses PIR for significant bandwidth savings.

Finally, we can also take special advantage of our PIR techniques to reduce the latency of our scheme even further. Normally when we retrieve a block from the ORAM, we read one element from each of the buckets to retrieve the single block we are actually interested in, discarding the other $\log n - 1$ blocks. With our “PIR bucket”, we do the same thing just without also retrieving the extra blocks that make up the individual buckets. However, we can apply PIR a second time to only retrieve the single block out of $\log n$ that we really want, reducing the bandwidth overhead to $O(1)$. Additionally, we can write back to the root bucket with $O(1)$ overhead, meaning that everything but the eviction can be done with optimal bandwidth complexity. Since the eviction is needed only to maintain the consistency of the tree, it is actually possible to “defer” several evictions to a later time and do only optimal read and add operations, which can be very attractive in a situation where you might have asymmetric bandwidth costs, i.e. expensive cellular network and cheap wifi at home.

Table 3 shows the worst-case bandwidth complexity and latency of our scheme and related work. We also show a “practical” worst-case bandwidth complexity which relies on moderately large blocks (100 kb) and realistically sized databases. We also show in figures 3(a) through 3(d) the relative bandwidth, time and monetary costs of Path-PIR and related work, as measured on Amazon EC2. Particularly, figure 3(d) shows how low our bandwidth is, even compared to Path ORAM, which is currently the most efficient ORAM but requires a more permissive model of non-constant client memory.
4 Applying ORAM to Robust Hidden Volumes

While ORAM can be used simply to outsource data while protecting a user’s access pattern, it is also a powerful tool with many widespread applications. We demonstrate this by showing how it can be used in a somewhat different way to provide robust hidden volume encryption, and interestingly how more restrictive adversarial models can allow for ORAMs with improved efficiency but that are still powerful enough to provide security for particular applications.

To understand what we mean by hidden volume encryption, it is necessary to introduce the current state of disk encryption. All major operating systems now support basic encrypted volumes natively, more and more users are taking advantage of these features. Additionally, there are open source software products, most prominently TrueCrypt [19], that provide more advanced solutions, beyond simply encrypting an entire disk.

One of the advanced features that TrueCrypt offers is a so-called “hidden volume”. Instead of a single encrypted volume, a user may optionally have two encrypted volumes. These volumes are encrypted with different keys (derived from passwords), and the user has the ability to plausibly deny the existence of the second volume. An adversary, knowing only the password to the first volume, cannot tell for sure whether there exists a second volume, let alone what its contents may be. Given the widespread use of encrypted disks, this is a very useful feature. If an adversary takes possession of an encrypted disk, they know that there is at least some data on that disk. They can...
then coerce the user to reveal their password used to encrypt the disk. With a hidden volume, the user can reveal the password to the first volume while withholding the password for the second. The adversary will not know whether the second volume exists, and therefore cannot be sure if there even is a second password for the user to reveal.

TrueCrypt accomplishes this by storing the second, “hidden” volume inside the free space of the first, “main” volume. Since the semantics of TrueCrypt guarantee that all free space in the encrypted volume will be filled with random data, and the encryption used is presumed to be indistinguishable from random, an adversary cannot tell if the blocks marked “free” in the main volume are actually free or if they contain encrypted data that is part of a hidden volume.

However, as already noticed by Czeskis et al. [5], TrueCrypt’s approach has a significant flaw: if the adversary has the ability to take multiple “snapshots” of the hard disk at different times, they can determine with high probability whether a hidden volume exists. Since disk encryption is specifically designed to protect against scenarios where the user loses control of their device, this is a major drawback. Requiring that an adversary has access to the machine and hard disk only one time is in many situations unrealistic. As a motivating example for why we consider multiple access to a hard disk a real threat, it is common for users to travel with their devices and lose direct possession of them on multiple occasions (checking bags, leaving them in a hotel room, etc.).

Consequently, we design a system whereby a user can plausibly deny the existence of a hidden volume even if the adversary has been able to take several snapshots of their disk and knows the password for the main volume. The reason that TrueCrypt does not maintain security against multiple snapshots is that it makes no attempt to hide the pattern of accesses that the user makes to the disk. That, combined with the fact that the hidden volume is stored separately from the main volume (in the free blocks of the main volume), gives the adversary a large advantage. An adversary can compare separate snapshots and see if a large number of “free” blocks have changed values. This would indicate that they are actually encrypted blocks that are part of a hidden volume, since they would otherwise not have a reason to change spontaneously.

These weaknesses lead us to our first observation: a system that is secure against multiple snapshots must make some attempt to hide the user’s access pattern. This is necessary for us to have some notion of “chosen plaintext” capability for the adversary. Otherwise, there would be many access patterns which could be easily distinguishable to them, e.g. writing to the same block in a hidden volume repeatedly versus writing to a variety of different blocks. As we know, we can hide access patterns using ORAM. Our approach will therefore be to store each encrypted volume as a separate ORAM, up to $max$ volumes. We then allow the user to specify how many volumes out of those $max$ they would like to use, which we denote with $w$. They specify $w$ passwords and we initialize the ORAMs with keys generated from those passwords.

The main idea will be that, whenever an operation is done on any volume, we will do a “dummy” operation on the other volumes, which does not change the contents of the disk, but in all respects is indistinguishable from a real operation to an adversary which does not have the key to that volume. This is easily done on an ORAM by reading a random block and writing back the same value unchanged. Now, an adversary cannot tell whether a particular operation was meant to be an access to a hidden volume or if it is simply a read operation in one of the volumes which they have the password to, since they look identical. The user can plausibly claim to not have the password to any volume (besides the first one, which is guaranteed to be in use) since there will always be a reasonable access pattern which would have produced the same sequence of disk accesses and does not include that volume.
At this point, it is interesting to note that for security we actually do not need an Oblivious RAM which hides reads which are done on the disk. This is because a disk drive does not leave evidence of reads. An adversary which can dump the contents of a drive cannot tell which blocks were read since the last time they looked at it, they can only see which ones were modified. Therefore, we can actually achieve the same level of security with a “write-only” Oblivious RAM.

We show that this type of Oblivious RAM is, conceptually, easier to achieve than a full-functionality one. In fact, it can be done with optimal, $O(1)$, overhead, while the most efficient ORAM previously, Path ORAM, only achieves $O(\log n)$ overhead. We accomplish this, at a high level, by moving from a tree-based structure to a flat, one-dimensional data store, while still maintaining access security. This is an independently interesting result, and it yet remains an open question whether our technique can be adapted to create an optimal ORAM which also hides reads or if there are lower bounds on fully-functional ORAMs which would disallow this.

Using our efficient write-only ORAM we show that our hidden volume encryption can be implemented in practice with relatively low overhead. Additionally, our scheme is particularly suited to the problem and can even be adapted to produce an optimized construction which stores the volumes interspersed with each other on the disk, reducing the amount of overhead added by each additional volume.

5 Publications

Our work on PIR in section 2 was published at Financial Cryptography 2013. Path-PIR, from section 3, was published at the Network and Distributed Systems Security Symposium 2014, where it received a distinguished paper award. Section 4, HIVE, is currently under submission to Computer and Communication Security 2014. Additionally, we have some related work on encrypted range search which is also under submission to CCS.

6 Proposed Work

There are three main topics I would like to investigate, in order to further the goal of a practical, usable Oblivious RAM. This work continues from our research on Path-PIR, hopefully further increasing its efficiency, and addresses some additional shortcomings of tree-based ORAMs. I would also like to investigate lower-bounds for ORAM and whether it is possible to have an $O(1)$ fully-functional ORAM or if there is something that is inherently “easier” about write-only ORAM.

**PIR with Path ORAM:** Our combined PIR-ORAM scheme, Path-PIR, is able to achieve similar asymptotic complexity to the most efficient ORAM currently known, Path-ORAM, but in practice average query response times are slower due to the fact that it is based on a less efficient ORAM scheme (in exchange for constant client memory). If, however, we were able to integrate our PIR techniques into Path ORAM, it would allow for an even more efficient solution that would not only be asymptotically better ($O(1)$), but more efficient in practice than all known ORAM constructions.

Path ORAM is also tree based, with each node in the tree being a bucket. We can use the same technique as before to achieve $O(1)$ retrieval, so the problem is only the eviction. However, it is non-trivial to adapt our technique to Path ORAM’s eviction. In the scheme we originally based
Path-PIR on, eviction is done in a predictable way, by removing one element from a parent node and inserting it into one of its children. However, in Path ORAM an entire path in the tree is downloaded and one or more of its elements are moved within that path, in such a way that blocks are moved as far as possible toward their respective leaves. This allows for the improved efficiency shown by the scheme, but it also makes it difficult to use PIR because a variable number of blocks are moved during each eviction. We cannot reveal how many blocks were moved, so in order to use PIR we would have to always issue the same number of queries regardless of the moves that actually need to be done. Since there can be up to $O(\log n)$ moves, and downloading the whole path only requires $O(\log n)$ bandwidth, we would gain no advantage using PIR this way.

However, if we could show that the minimum number of moves needed in order to guarantee the consistency of the tree is actually bounded less than $O(\log n)$, say by a small constant, then we could use PIR again. So far we have done experiments that suggest this may be the case, but there is significant work that still needs to be done in proving the soundness of this approach.

**Resizable ORAM:** As stated before, one of the downsides to newer ORAM constructions is that they are no longer easily resizable. Previously, one could always resize the data structure when it was being shuffled, but now that the “shuffling” happens a little bit at a time in every step, there is no longer a point when it can be done without ruining the worst-case complexity. This can be a significant drawback, especially when considering the use of ORAM for outsourcing data. If a user has to pay for the maximum amount of storage they will ever use, even when they are currently using far less, it would be a very unattractive option (especially considering one of the main advantages of the cloud is elasticity).

While it may be relatively simple to double the size of the ORAM by simply adding another level of leaf nodes, it remains an interesting question whether the size can be controlled in a more fine grained manner. It may not always be desirable to double the size of the storage, but rather slowly increase it. Also, some tree based ORAMs require leaf nodes and interior nodes to be different sizes, which would be ruined if levels are added on the fly (since leaf nodes would become interior nodes). In contrast with expanding, there is no trivial way to decrease the size of an ORAM.

Interestingly, it may be possible to decrease the size of an ORAM that is encrypted with a homomorphic encryption scheme, like Path-PIR. If there were few enough elements in the tree that it could be guaranteed that combining the leaf nodes with their parents would not cause an overflow, the client could provide to the server a permutation of the bucket slots for each leaf node that would guarantee they could be additively “combined” with their parent and sibling without overwriting any real data blocks. This requires a relatively large amount of data from the client, but it is independent of the block size, $\ell$, since it is just permutation information. I would like to continue investigating this technique and other, more general solutions to resizing these modern Oblivious RAMs.

**$O(1)$ ORAM:** Our work from section 4 shows that, with logarithmic client memory, it is possible to achieve $O(1)$ “write-only” Oblivious RAM. There has been little work on lower bounds for Oblivious RAM, and it is currently only known that for constant client memory it must require $\Omega(\log n \log \log n)$ overhead [2]. For even small amounts of client memory (logarithmic), only the information theoretic lower bound of $\Omega(\log n + \ell)$ is known (log $n$ bits for the block index and $\ell$ bits for the block itself). All tree-based schemes assume that $\ell > \log n$, and with such an assumption this bound becomes trivial ($\Omega(1)$ overhead).

I would like to investigate lower bounds under logarithmic client memory in order to determine whether there is separation between write-only ORAM and fully-functional ORAM. Or,
conversely, whether we can construct an $O(1)$ Oblivious RAM using similar techniques as our write-only scheme.

### 6.1 Timeline

<table>
<thead>
<tr>
<th>Date Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 2014</td>
<td>Dissertation proposal</td>
</tr>
<tr>
<td>September - March 2015</td>
<td>Continue proposed work</td>
</tr>
<tr>
<td>March - June 2015</td>
<td>Write dissertation</td>
</tr>
<tr>
<td>July 2015</td>
<td>Defend dissertation</td>
</tr>
</tbody>
</table>

### References


