# Protecting Private Data in the Cloud: A Path Oblivious RAM Protocol

Ethan Zou and Nathan Wolfe

November 2014

**Abstract**

*We live in a world where our personal data are both valuable and vulnerable to misappropriation through exploitation of security vulnerabilities in online services. For instance, Dropbox, a popular cloud storage tool, has certain security flaws that can be exploited to compromise a user's data, one of which being that a user's access pattern is unprotected. We have thus created an implementation of Path Oblivious RAM (Path ORAM) for Dropbox users to obfuscate path access information to patch this vulnerability. This implementation differs significantly from the standard usage of Path ORAM, in that we introduce several innovations, including a dynamically growing and shrinking tree architecture, multi-block fetching, block packing and the possibility for multi-client use.*

## I. Introduction

Nowadays, with many people possessing a multitude of devices, many of which have limited local storage, it is popular to store files in the cloud. Among the most popular cloud storage tools is Dropbox, which is compatible with computers and mobile devices, and allows users to store up to 2 GB of files for free (additional storage is available for a monthly fee). For most users, Dropbox presents adequate security; all files are encrypted when stored on their servers. Nevertheless some will rightfully find certain issues with their system. One is that Dropbox can see which files you edit. This, known as the user's access pattern, is a key piece of information, which Dropbox could potentially gather. Of course, Dropbox itself is trustworthy, but nowadays the US government has the power to request data from web services, and the access pattern may be included. While standard encryption offers some protection, it does nothing to hide a user's access pattern.

Previous works have shown the importance in protecting access pattern. Memory access patterns can leak information such as a program's control flow, which concerns conditional branches of the program and the order of commands executed [11]. Also, M.S. Islam et al. have shown how, from an encrypted email repository, adversaries can infer 80% of search queries simply from access patterns [3].

Oblivious RAM (ORAM) is designed specifically to obfuscate what files in a database are being edited. This obfuscation of the user's access pattern provides a critical additional layer of security. Thus, in our research project we have implemented an ORAM system for use with Dropbox. Along the way, we have discovered various innovations and optimizations that are unique to the Dropbox usage scenario. First, we have designed a system where ORAM on Dropbox can be used between multiple user computers. We also propose the optimizations of multi-block fetching, which improves performance by 51.9%, and block packing, which saves 50-70% storage overhead. Finally, we propose a dynamically growing and shrinking tree, providing us with the necessary storage size flexibility to work with Dropbox.

In this paper, we will first provide some background on ORAM. We will then explain the system architecture behind our ORAM implementation. Then, after explaining our various optimizations and improvements to the base system, we will present some evaluations performed on the system to validate its effectiveness and assess its performance.

# II. Background

## I. Oblvious RAM

First, what is ORAM? Imagine that you have a table with papers lying on it in a common room at your office building. Occasionally you want to read or write on these papers. The problem is that your coworkers can see when you read or write on a sheet, when you really want to keep which one you are working on a secret. One good option to hide which you are working on is to take all of the papers into your private office, do what you want, and then bring them all back out to the table again. This is what we call naive Oblivious RAM (ORAM). Your coworkers are analogous to adversaries, the table outside is untrusted storage, and your own office is trusted. The sheets of paper represent blocks of data, and working on a sheet is analogous to accessing it, that is, reading or editing it. The point of the system we just described, and of any ORAM system, is to hide your access pattern: which sheet you are working on. This naive system works well to achieve this goal, but is obviously very inefficient. (Why even keep the papers outside if you bring them into your own room all the time anyway?)

In the past three decades, researchers have strived to improve the efficiency and reduce the trusted storage requirement of ORAM [1, 2, 4, 7, 8]. A breakthrough in implementation is achieved through Path ORAM [9]. Path ORAM's performance and trusted storage requirement are both logarithmic in its capacity. Furthermore, its algorithm is extremely simple and suitable for real world implementation. Immediately after its proposal in 2012, Path ORAM gained popularity in many applications due to its simplicity and efficiency. In this work, we adapted it for use with Dropbox.

## II. Path Oblvious RAM

To make the Oblivious RAM system more efficient, researchers introduced Path ORAM. In this system, blocks of data are arranged into a binary tree, with a few blocks at each node of the tree. (A binary tree, pictured in **Figure 2-1**, is a tree in which each node has up to two children.) Instead of accessing all blocks in order to keep a specific one secret, we can simply access all blocks along the target's path. "Accessing" in this context means fetching a block from untrusted storage and placing it in memory to be worked on. Work on a specific block is still hidden because a group of blocks are accessed together, but the groups are much smaller, improving speed.

We summarize how Path ORAM works here. The reader can refer to [9] for more details. The tree is a binary tree data structure of height $L$ and $2^L$ leaves stored on the server as a one-dimensional array. The levels are numbered 0 to $L$, where level 0 is the root, level $L$ contains the leaves (in a complete tree), and levels in between contain non-leaf nodes. Each node in the binary tree is called a bucket, and contains $z$ blocks. If a bucket contains less than $z$ "real" blocks, the remainder are composed of dummy blocks, which are indistinguishable from real blocks because buckets are stored as encrypted fixed-size arrays.

On the client side, we construct a data structure called the **stash**, which stores a small subset of the data blocks, and a **position map** that assigns each block to a path in the tree. Every path is uniquely defined as leading from the root to one of the $2^L$ leaf buckets. Importantly, mapping is random, allowing for multiple unrelated blocks to be mapped to one leaf. This is shown in **Figure 2-1**: for example, both blocks B and E are mapped to leaf 1, and are required to reside on the path defined by this leaf (referred to as path 1). The position map is updated as blocks are accessed and their leaf assignments changed. This is critical because Path ORAM maintains the invariant that if the block is not on its assigned path, it is in the stash (if it is not, there is a bug).

Reading/writing a block consists of several steps. First, we consult the position map to determine what path the target block is on. Then, we fetch the blocks on that path from the tree. Based on the invariant just mentioned, it should be among those fetched blocks or in the stash. We can then securely access the target block. After making any necessary changes to the block we assign it to a different path using the position map. Finally we take all blocks originally fetched and write them back into the

tree.

In the past many papers have focused on Path ORAM systems and their optimizations. However, this is the first applying Path ORAM directly for use with Dropbox, and the first to explore optimizations unique to the Dropbox usage case.

## III. System Architecture

We introduce the general organization of our design and several changes to the basic implementation to better suit Dropbox.

### I. Big Picture

The overall interface of our design consists of three parts: the client, the Path ORAM implementation and Dropbox. Instead of the client directly writing and reading files to the Dropbox folder, we implemented the ORAM as an intermediary controller between the two. So a user will directly interact with the User File System (See Section 3.2), which in turn interacts with the ORAM controller to read and write files. The ORAM controller interacts with Dropbox by having the buckets of the tree written to the Dropbox folder. The Dropbox service will then sync the contents to the cloud. When reading a file, the bucket files are downloaded. As we will discuss in Section 3.3, a system with multiple computers accessing the same files requires the stash, position map, and other dictionaries to be written to Dropbox as well, which can be downloaded by other computers in the system. These structures which are normally kept in secure storage have to be encrypted with a secret key.

### II. User File System

Dropbox's baseline functions allow users to read and write files to a remote server in the cloud. Thus, we needed a way for our ORAM system to support the reading and writing of files of various sizes. Our solution was implementing the User File System, which enables the writing of files of different sizes.

The User File System interacts with the ORAM controller and provides the users with an interface to read, update and delete files of arbitrary size. The User File System consists of several dictionaries which hold vital metadata. Its main feature is to partition files into chunks, whose size is a parameter we call **segment size**. These chunks are then each written to or read from the ORAM.

To write a file, we split it up into subfiles each containing a number of bytes equal to (or less than, in the case of the last subfile) the segment size. We then assign each subfile a unique segment ID (stored in a dictionary) that allows us to be able to track the individual segments when we want to read the file again. Then, we write each file segment to the ORAM and update another dictionary with the number of partitioned pieces in that particular file. When reading a file, we look up the file name in a dictionary to retrieve the number of segments in that file. Then, we obtain the individual segment IDs of each data segment and fetch them from the ORAM. The data is then stitched together and returned to the user. Deleting a file works similarly to reading. After looking up the segment IDs of the file fragments, we simply delete those data from the ORAM and delete their corresponding information from the two metadata dictionaries.

### III. Multi-Computer

One of the major reasons cloud storage programs such as Dropbox are so popular is that a user can access and change files on one computer, and then move to another computer to access the same files. In order to support this feature, we added functionalities to the file systems.

When a user "logs off" from the computer, the dictionaries in the User File System, the position map, and the stash are written to the Dropbox folder, and then synced to the remote server. An important implication is that these files are encrypted and padded to a constant file size when written to Dropbox to ensure security and prevent an adversary from gaining information from file size or access pattern. When the user wants to access his files from another computer, he/she downloads these files from the server to reconstruct the User File System and ORAM metadata structures.

## IV. Initial Evaluations

Our current implementation is significantly slower than directly using Dropbox without ORAM. Thus, the implementation has some apparent drawbacks. The major problem comes from partitioning an immensely large file into millions of segments (based on the assigned segment size), which means that reading or writing one large file takes a very long time. Along with this, a static tree size is a problem when using Dropbox because we don't want space to be wasted (more space equals more money). With these issues in mind, we sought to develop optimizations to improve upon our initial implementation.

# IV. Optimizations

In this section we provide detailed explanations and motivation for the new optimizations we have implemented: multi-block fetching, block packing and a dynamically growing and shrinking tree size.

## I. Multi-Block Fetching

One idea to improve performance dramatically is to fetch multiple data blocks in each access, thus cutting down on the total number of accesses required. Unfortunately, usually when using ORAM, it is unclear how data blocks are grouped outside, so it is hard to know which groups of blocks to fetch together. However, in our implementation of ORAM with Dropbox, the data segments stored can be grouped in a defined way, with each grouping corresponding to a particular file. Thus we can cut down on the number of tree accesses we perform by putting segments from the same file on the same path in the tree and fetching them at the same time. The technique is as simple as grouping segments from each file into n-tuples, mapping them all to the same path, and then fetching them all at the same time. Theoretically this should reduce the number of accesses needed by a factor of n. Unfortunately, however, grouping a great number of segments together in the same path produces inefficiency. Any individual path may become congested, slowing down the operation. Through evaluations of file access speed we have managed to find an optimal value for n for this grouping process.

## II. Block Packing

One way to speed up the ORAM operations is to cut down on the number of partitions of a file. In order to do that, we need to use a large segment size. However, larger segment sizes hurt small files because a tremendous amount of space is wasted, and it takes longer to read small files.

With this in mind, our approach to improving this aspect of the ORAM was to implement block packing, which essentially packs more than one file segment together into one block. The motivation behind this approach comes from the idea that a small file does not necessarily need to be the only file present in a block. Additionally, leftover data that does not divide evenly into the segment size can also be packed into the empty spaces of a block. By doing this, space is saved and not wasted, resulting in a smaller tree and smaller path lengths (see Section 4.4), and consequently reduces file access time (though only slightly).

Block packing works as follows. We add two more dictionaries to the User File System: one holds the amount of space left in a block and the other holds the start and end offsets of a certain file segment inside of a block. We write a file of an arbitrary size. The "full" segments of data (take up an entire block) are written normally to the ORAM. We then look for available space in a block for the very last segment (which, in the case of a small file will be the entire file). If we find a block that already has some data in it and that has sufficient excess space, we append the final segment to the end of the already existing data, and update the dictionaries accordingly. If we do not find an available block, we write the data to an empty block and add the ID of that block to the dictionary with its amount of empty space. When we want to read the file, all the segments except the last are read normally. We find the position of the last segment in the block and read out the specific portion of the data.

Deleting a file becomes slightly more complicated. All file segments except the last one

are deleted normally. When deleting the last segment of a file, we run into the problem of there possibly being a "hole" of no data inside a block. Our solution was to shift the data after the deleted segment as far up in the block as possible so that there are no gaps in the middle of a block. We then update the dictionaries for the shifted segments of data.

## III. Dynamic Growing and Shrinking Tree

A normal ORAM implementation involves a storage space that is fixed in size (e.g., main memory or hard drive of a computer). However, Dropbox presents the unique problem of a dynamically sized storage space. First, users may wish to store sensitive files in an ORAM system alongside less important files stored unencrypted in Dropbox. By supporting resizing of the tree we support this usage scenario. Second, Dropbox allows users to increase the size of their storage by either paying or sharing the service with friends. By supporting tree resizing we allow users to change their storage space without having to create a new ORAM library each time. Thus, resizing provides convenience to the user in a way unique to this project.

The first problem we encounter is to determine when to resize. Previous work suggested that the optimal utilization (meaning the fraction data block slots that are used) is around 50% [5].

Our implementation supports both growing and shrinking. Both functions are triggered based on current utilization,. When the utilization goes above a certain threshold we grow the tree until the utilization goes back down to a target level. When it goes below a certain threshold we shrink (to save Dropbox space), with the utilization going back up to a target level.

Growing the tree works as follows. First, we add the correct number of buckets to the leaf end of the tree. Obviously this reveals the information that we are growing, and that the total size of our files stored has gone beyond some threshold, but that information is both non-critical and practically unavoidable to leak. We then correct all leaves in both the position map and the stash in order to conform to the new leaf numbers. Nodes with new children are invalidated as leaves, so records pointing to these nodes are reassigned to one of the children.

Shrinking works similarly; first we remove the correct number of buckets from the leaf end of the tree, dumping their contents into the stash. Again, this leaks the information that our total file size must have decreased. We then correct all leaves in the position map and the stash by truncating the path so that it lines up with the correct leaf, and leave the ones in the tree to sort out later.

## V. Evaluations

In this section, we evaluate the performance of our optimizations.

I. Mehthodology

II. Optimal Segment Size

III. Multi-Block Fetching

IV. Block Packing

V. Dynamic Growing and Shrinking

VI. Total OptimizationImpact

## References


[1] Goldreich, O. (1987). Towards a theory of software protection and simulation by oblivious RAMs. *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, 182-194.

[2] Goodrich, M. T., & Mitzenmacher, M. (2011). Privacy-preserving access of outsourced data via oblivious RAM simulation. *Proceedings of the 38th International Conference on Automata, Languages and Programming, Part II*, 576-587.

[3] Islam, M. S., Kuzu, M., & Kantarcioglu, M. (2012). Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. *NDSS*.

[4] Ostrovsky, R. (1990). Efficient computation on oblivious RAMs. *Proceedings of the*

*Twenty-second Annual ACM Symposium on Theory of Computing*, 514-523.

[5] Ren, L., Yu, X., Fletcher, C., van Dijk, M., & Devadas, S. (2013). Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors [PDF]. *International Symposium on Computer Architecture*.

[6] Roselli, D., Lorch, J. R., & Anderson, T. E. (2000). A Comparison of File System Workloads. *2000 USENIX Annual Technical Conference*.

[7] Shi, E., Chan, T.-H. H., Stefanov, E., & Li, M. (2011). Oblivious RAM with o((logn)3) worst-case cost. *Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security*, 197-214.

[8] Stefanov, E., Shi, E., & Song, D. (2012). Towards practical oblivious RAM. *NDSS*.

[9] Stefanov, E., van Dijk, M., Shi, E., Chan, T.-H. H., Fletcher, C., Ren, L., Yu, X., & Devadas, S. (2013). Path ORAM: An Extremely Simple Oblivious RAM Protocol [PDF]. *Proceedings of the 20th Computer and Communication Security Conference*.

[10] Yu, X., Ren, L., Fletcher, C., Kwon, A., van Dijk, M., & Devadas, S. (2014). Enhancing Oblivious RAM Performance Using Dynamic Prefetching. *IACR Cryptology ePrint Archive*.

[11] Zhuang, X., Zhang, T., & Pande, S. (2004). Hide: An infrastructure for efficiently protecting information leakage on the address bus. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 72-84.