Optimizations and Discoveries Regarding Constant-Communication Oblivious RAM

Akiva Gordon, Krishna Suraj

Abstract

In today's world, an ever-increasing amount of data is being stored online in the "cloud" - external, untrusted servers. Considering the importance of such data to users, businesses, and organizations, it is necessary to protect data from mishandling and exploitation from those servers. While current encryption schemes can protect the actual data on a server from being read, a user's access pattern to files on a server is vulnerable. Oblivious RAM (ORAM) is a cryptographic primitive that hides a user's access pattern metadata from untrusted servers. In this work, we discuss and analyze various ORAMs. We first provide an explanation of the various tree-based ORAM algorithms, including Constant Communication ORAM (C-ORAM), a constant-bandwidth, high-efficiency algorithm. We then provide an experimental analysis of C-ORAM using our implementation of the algorithm in Python 2.7. Using data from our experimental analysis, we determine runtime as well as optimal bucket size and eviction frequency based on many accesses to our ORAM implementation.

1 Introduction

Since the advent of the "internet age", the number of files and data being created grows exponentially every year. Users store an ever-increasing amount of data on the internet using cloud services such as Google Drive and Dropbox. Even more significantly, thousands of businesses store private data on the cloud using services such as Amazon Web Servers (AWS). With an ever-increasing amount of data being created and stored in the cloud, a user's privacy and anonymity in the digital world is becoming more important than ever before. To protect private data and enterprise secrets, users and businesses are increasingly concerned with the threat of data theft.

Important data in the cloud should be encrypted, such as in online transfers (e.g., PayPal), and current encryption schemes do a good job of protecting actual data. However, a user's metadata created when accessing and operating on data is still unprotected. For example, when a business stores encrypted data on external or rented servers (e.g., Amazon Web Servers) the hosting server can still see which files are being accessed, the frequency of accesses, and the time of these accesses. These metadata, or access patterns, are important to protect because otherwise an untrusted server may deduce information about the files themselves. For example, if a certain file is accessed more frequently, the server may deduce that they it is an important file. Also, if certain files are accessed very frequently before a certain date (e.g., tax day), the server may conclude that those files serve a certain purpose (e.g. a user's private tax returns).

Oblivious RAM (ORAM) is a cryptographic primitive that enables a user to store data on an untrusted server while maintaining data obliviousness - that is, the server cannot gain information regarding the user's access pattern of their files.

ORAM preserves the confidentiality of these access patterns, allowing the user to conceal their activity from the server. Thus, ORAM protects the user from any potential metadata analysis by the server, creating a higher security storage system.

In this work, we examine various recent ORAM schemes. Path ORAM [7] and Ring ORAM [6] are tree-based ORAMs that have a bandwidth with an asymptotic efficiency of $O(\log N)$. Onion ORAM [2] breaks the $O(\log N)$ barrier by using server computation to achieve constant bandwidth from and to the client. However, the server computation is extremely inefficient. Constant Communication ORAM (C-ORAM) [5] is a recent ORAM scheme using server computation that builds upon Onion ORAM. C-ORAM reduces the overall bandwidth of Onion ORAM by introducing novel eviction techniques that eliminate the need for the multiple encryption layers used in Onion ORAM. In this work, we perform an experimental analysis on C-ORAM using a Python 2.7 implementation. Using this implementation, we conduct runtime tests using multiple access cycles on the ORAM as well as determining the optimal eviction frequency and bucket size for efficient C-ORAM server computation.

2 Previous work

2.1 Previous tree-based ORAM schemes

Initially proposed by Goldreich and Ostrovsky [3] [4], Oblivious RAM has long remained in the theoretical domain, with bandwidth being a limiting factor preventing the practical adoption of ORAM. Path ORAM demonstrated the use of a tree data structure for ORAM, which achieved an asymptotic efficiency on the order of

$\log N$ - a significant step. Path ORAM introduced the use of a binary tree to store an ORAM structure. Units of data called "blocks" are stored in "buckets" - the nodes of the binary tree. Each bucket occupies one whole node of the binary tree. Each data block has a unique path to a leaf node on the tree. A data access in Path ORAM consists of a request to the position map, which stores a list of data blocks and their corresponding paths. Then, the user pulls the entire contents of the path into the "client space." The client space is assumed to be fully secure, which means that it cannot be seen or modified by the server. The selected data block is operated on using a read/write operation. All other blocks are placed back on their original paths, but as far down towards the leaves as possible. The accessed block is assigned a new path to ensure that the server cannot guess which path the block is on to maintain obliviousness, and placed at the bucket which is the lowest intersection on the tree of the old and new paths.

Ring ORAM, another tree-based ORAM, is an optimization of Path ORAM. One improvement is the reduction of bandwidth use during an access. If a tree has l levels and each bucket has z blocks, Ring ORAM pulls only l blocks to the client, rather than $l * z$ blocks in Path ORAM. It does this by pulling only one block from each bucket on the path, rather than all blocks on a path. When traversing a path during an access, if the desired block is present in the current bucket, it is retrieved to the client and replaced with a dummy block - a block with meaningless, randomly generated data. If not, a reserved "dummy" block, which is intended for the purpose of retrieval is sent back. The dummy blocks are discarded on the client side. Figure 1 shows the process of selecting one block from each bucket in Ring ORAM.

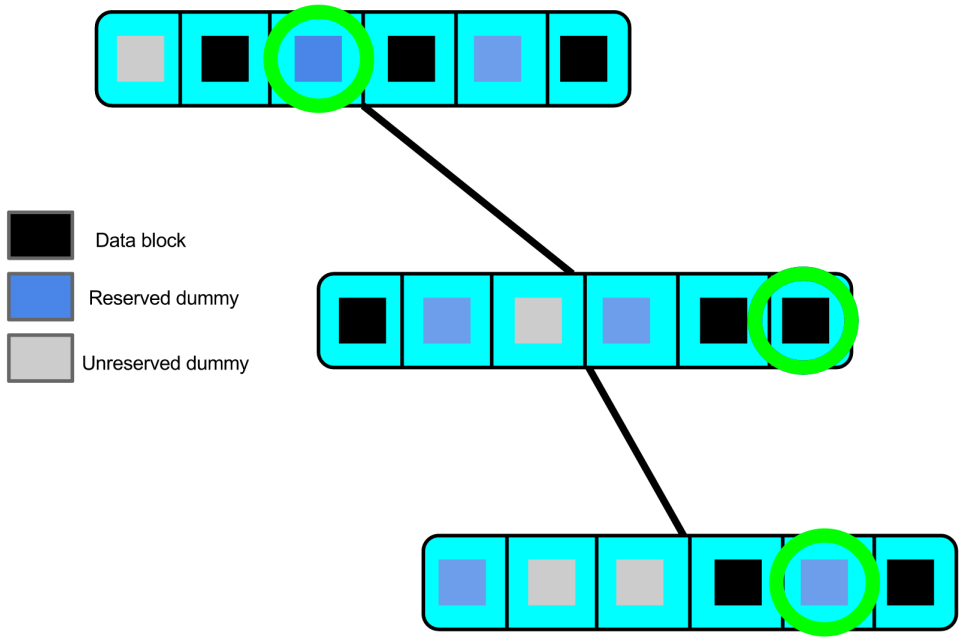

Figure 1: Ring ORAM access. The circled blocks are selected. The buckets which do not contain the target block (levels 1 and 3) send back a reserved dummy block

Ring ORAM also proposes two changes to Path ORAM's eviction process. First, eviction only occurs after every A accesses, thus, evictions may occur less frequently then in Path ORAM. This reduces bandwidth use, increasing efficiency. Second, instead of replacing blocks in the tree along the path they were retrieved from, they are replaced along a path chosen through reverse lexicographic order. Although A and the path of eviction are public, they are deterministic and therefore independent of the block or the action taken, so they do not compromise the security of the system. Figure 2 shows the process of selecting eviction paths in Ring ORAM, Onion ORAM, and C-ORAM.

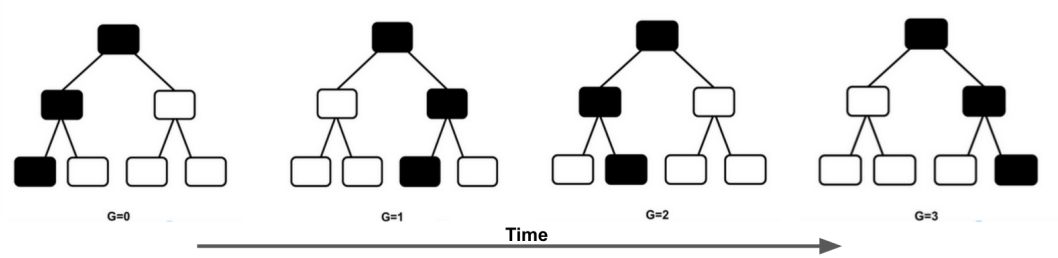

Figure 2: Reverse lexicographical order balances the tree through standardized evictions.

2.2 Onion ORAM

Onion ORAM implemented, using additively homomorphic encryption, the idea of having the server conduct some computations for the user rather than all computation being conducted on the user's end. Although earlier works have used server computation, Onion ORAM is the first to achieve constant bandwidth. all data is encrypted and shuffled on the server, so it is very counterintuitive that the server can do any useful work to access or retrieve the files. The server computes on an encrypted bit string and returns a number of blocks, one "real" block and the rest "dummy" blocks so the server cannot know which block the user wants. The dummy blocks are discarded. This is called a Private Information Retrieval. To conduct a private information retrieval, the client first extracts the information to what path the desired data block is on using the position map. Using the path, the client sends an encrypted bit string to the server which the server computes on the eviction path to return one block from each bucket. The server will return dummy blocks from all buckets except for the bucket where the desired data block is, when it will return the data block itself. On the client side, the dummy blocks are discarded and the data block is subject to either a read or write operation. The block is

then pushed back on to the Onion ORAM tree at the root node.

2.3 Onion ORAM Eviction

Evictions in Onion ORAM do not happen every access to the block, but periodically. The parameter A describes this frequency. The eviction step of Onion ORAM consists of all blocks on a certain eviction path being pushed all the way to the leaf nodes of the tree. Each step of this eviction, a new layer of additively homomorphic encryption must be added to the data as it is additively merged with the data below it. This, after many operations, will result in a very long cipher text and a very high bandwidth cost. Thus, the encryption bottleneck is very costly in terms of the server's computation efficiency. After all the data is evicted to the leaf, the non-leaf parent blocks are cleared, resulting in an empty path save the leaves. Onion ORAM uses a costly polynomial encryption scheme - Damgard Jurik [1]. Encryption in Onion ORAM is considered to be a bottleneck, resulting in a poor computational efficiency although a constant bandwidth efficiency successfully breaks the $O(\log N)$ bound previously suggested to be the lower bound for ORAM bandwidth efficiency.

3 C-ORAM

Constant communication ORAM (C-ORAM), proposed by Moataz, Mayberry, and Blass seeks to replace the homomorphic eviction routine in Onion ORAM with a cheaper eviction sequence that replaces the homomorphic multiplications of Onion ORAM while maintaining security. This removes the need for Onion ORAM's layers of encryption and reduces the minimum block size and server computation cost.

We start by introducing the C-ORAM process and then provide an experimental analysis.

3.1 Accesses

To perform a data access in C-ORAM, the client first retrieves a location tag for the block from the position map (an additional data structure that stores the locations of the specific blocks. This tag defines the path in which the unique data block is located on. Then, the client conducts a Private Information Retrieval applied to each bucket on the path. A data retrieval is performed on every bucket on the specific path to maintain obliviousness - this way, the server does not know which specific data block is being accessed. The client thus retrieves the data block and can perform read or write operations on it in the secure client space. Afterwards, the client can perform a private write into the tree, placing the data block into the root of the C-ORAM tree. C-ORAM accesses are identical to Onion ORAM accesses.

3.2 Evictions

In C-ORAM, an eviction only happens every A operations, where A is a parameter. A path to evict along is, like Onion ORAM, chosen following reverse lexicographical order. First, the root bucket, which is assumed to be empty (e.g.. filled with $E(0)$'s (encryptions of zeros), is accessed and written to with encrypted data. This is a normal private write operation.

Constant Communication ORAM differs from Onion ORAM in its eviction step. C-ORAM eviction occurs when the data in each bucket on the selected path merges its data with its two child buckets. These merges are called Oblivious

Merges and are detailed in the following section. Only one of these buckets will be on the desired path, so the "destination" bucket's "sibling" will have data blocks that are not intended for it but for the destination bucket. Likewise, the destination bucket will have data blocks from the parent bucket that were meant for its sibling bucket. These blocks are called "noisy" blocks and are functionally useless and disposable since an exact copy of the data exists elsewhere. After every bucket on the path has been merged with its two children, the leaf buckets evicted to will have all the data, so the parent buckets along the eviction path are cleared - e.g., filled with $E(0)$'s. This completes an eviction in C-ORAM. Before a C-ORAM eviction, the sibling buckets of the eviction path are given to be empty except for the leaves. After an eviction, all the non-leaf buckets on the path are empty while the sibling buckets are not. After a certain number of operations, the leaf buckets are also downloaded and its noisy blocks are cleared (to prevent overflow). Figure 3 shows the process of eviction.

3.3 Oblivious Mergings

Oblivious Merging is a method that combines two given buckets in a secure method using server computation. It permutes two buckets so that they are obliviously "lined up" and combined into a single bucket. Rather that just appending the data in one bucket to the end of another bucket, oblivious merging saves spaces by"overlapping" the two buckets. Since noisy blocks are functionally useless, the buckets are permuted so that real data blocks in one bucket line up with either noisy blocks in the other bucket, and vice-versa. For example, in a merging of buckets A and B, the ith position in bucket A - given to be a real block - will line up with either

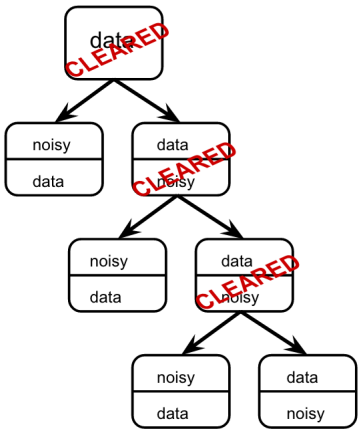

Figure 3: Shows the eviction pattern down a particular path in C-ORAM. Although all the data being pushed down to children buckets by parent buckets are identical, noisy blocks on the eviction path are relevant data to their sibling buckets, and vice versa.

a noisy block or an zero-encrypted block in position i in bucket B. Thus, when the buckets are simply combined, the real blocks are prioritized and preserved and the noisy or zero-encrypted blocks with which they line up with are discarded, losing no information.

A concern for C-ORAM is that during repeated oblivious merging, the data and noisy blocks will eventually exceed the storage space of the bucket and the bucket will "overflow." Thus, it is necessary to determine a minimum size bound at which the bucket does not overflow. Mayberry et al. devises a theoretical proof for this in their C-ORAM proposition work. We address the bound using experimental methods in our tests, outlined below in our contributions.

4 Our Contributions

In our project, we explain intuitively why the oblivious merging in C-ORAM is secure. We also argue that during oblivious merging, only a single block needs to be permuted and aligned with the other. Also, we tested the merging eviction protocol for consistency and loss of data:

4.1 Oblivious Merging Security Explanation

A main concern with oblivious merging is the security of the permutation conducted by the server to obliviously line up the two buckets. Since C-ORAM's security proof is not intuitive, we provide an intuitive explanation: if bucket A is randomly permuted and bucket B is permuted to fill the spaces in bucket A, bucket B's permutation must also be secure. To elaborate: considering Figure 4, the left bucket is assumed to be randomly permuted - that is, the server does not know the locations of specific real, noisy, or $E(0)$ blocks in the bucket. The right bucket is permuted so that real blocks line up with $E(0)$ blocks to prevent a loss of information. Also, noisy blocks are lined up with other noisy blocks to keep the number of noisy blocks in the merged bucket low. This second permutation must be secure and as random as the orientation of the left bucket because it is dependent on the left bucket. Thus, intuitively, we come to an understanding that the oblivious merge operation is secure. Figure 4 visually diagrams the process of oblivious merging.

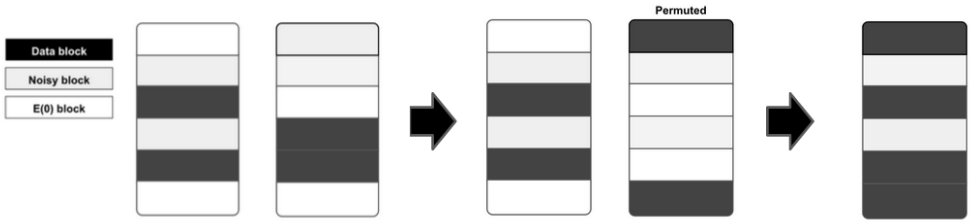

Figure 4: Two buckets are permuted so "real" data blocks (in black) are lined up with $E(0)$ blocks (in white) and noisy blocks (grey) are lined up with other noisy blocks. Thus, merging the two blocks occurs without the loss of real data blocks while keeping the number of noisy blocks low

4.2 Single-bucket permutation

We argue that during an oblivious merge operation, it is only necessary to permute one bucket. In figure 4, only the right bucket is permuted and the left bucket stays in its original ordering. This is secure because the left bucket is assumed to be a random permutation - the server cannot determine which blocks are noisy, real, or $E(0)$. Thus, if the left bucket is already secure from the server, it is only necessary to permute the right block. This is a benefit for computational efficiency as it reduces the number of operations in oblivious merging by a factor of 2 - an optimization of server computation.

4.3 Experimental Results

Using Python 2.7, we created a simulation of C-ORAM's eviction process to study certain parameters. This program was able to test the time performance and efficiency of C-ORAM over many accesses. It also included the ability to set certain parameters such as bucket size and eviction frequency. We ran tests with ORAM

trees ranging from 10 to 15 layers in depth with bucket sizes ranging from 500 to 10000 blocks. The computer used was a Lenovo Thinkpad (SL500 2007) running Windows Vista. It had 4.00 gB of RAM, and a 2.00 gHz Intel Core2 Duo CPU. The programs were interpreted with Python 2.7. It is important to note that our program was a simulation - very useful for testing and determining the optimal values of certain parameters and visualizing the general performance trend, but not necessarily with the same performance as a full C-ORAM product for users. Specifically, blocks were not encrypted (to save bandwidth) and accuracy tests were performed through simple reads of the storage tree rather than through the Merkel tree structure used in most ORAMs. However, this simulation allowed us to gain useful insights into certain aspects of the algorithm's performance.

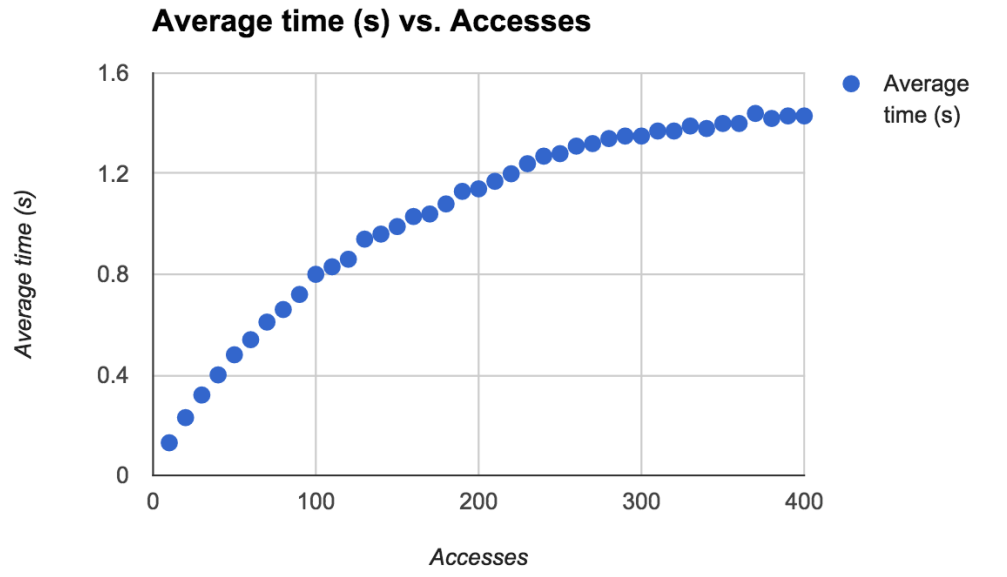

Figure 5: Graph 1: Evaluation of runtime of program based on blocks already in tree

In our tests, it became clear that the speed of the algorithm is asymptotically dependent on the number of blocks already in the tree. This does technically support the constant bandwidth requirement, seeming to be dependent on the constant bucket size, rather than the number of blocks added. However, it will always be true that the program will shuffle and re-align buckets more quickly when fewer real blocks are present. For instance, for a tree with 13 layers and $z = 4000$, through our testing it became evident that the average eviction time reaches an asymptotic limit of ≈ 1.4 seconds after about 400 accesses.

In a test for tree stability, blocks were repeatedly added to the tree via eviction until a bucket overflowed. Results from our experimental test are in Figure **??**. We learned that bucket overflow errors occurred quite quickly if the number of blocks added per eviction exceeded approximately 10 % of the bucket size. This presents a minor problem for storing many blocks, as frequent accesses will necessitate the need for evictions of many blocks at once. To correct for this issue, it is necessary to either evict at about every $\frac{z}{100}$ accesses, or to divide evictions into several smaller evictions of fewer blocks to distribute the stress among multiple paths of the tree.

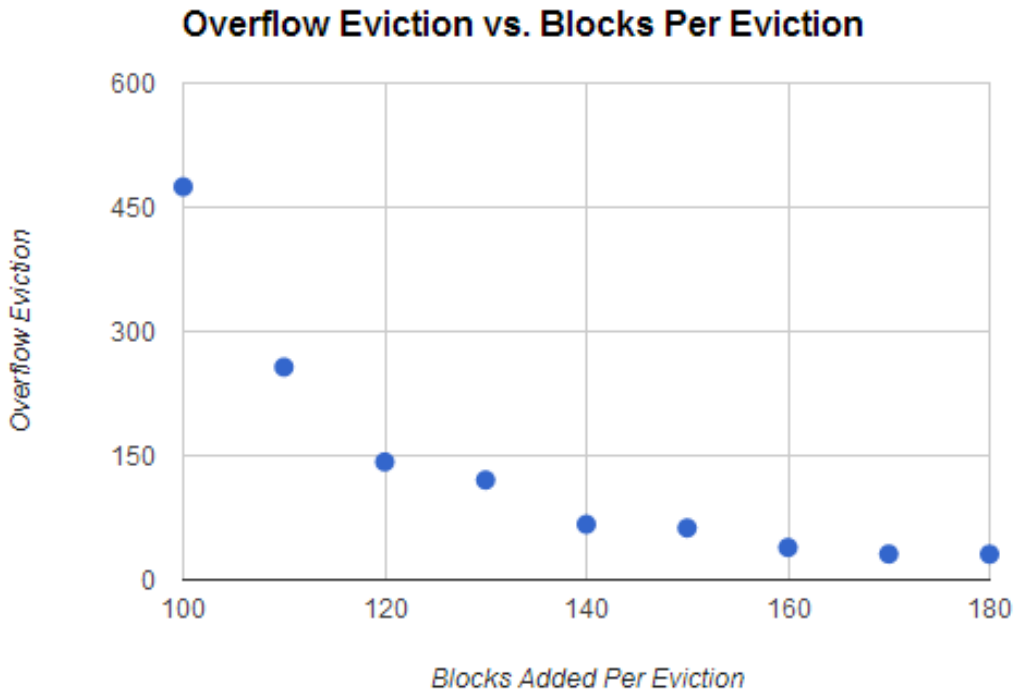

Figure 6: Graph 2: Evaluation of overflow of tree when adding blocks without removal

In addition, noisy blocks must be cleaned from the tree as they will accumulate in the leaves if not dealt with otherwise. This can cause bucket overflows if too many blocks are evicted through the tree. In our protocol, we cleared the leaf of the eviction path to prevent the tree from filling with noisy blocks, which reduced this overflow rate. A leaf clear is, in other words, a replacement of all the blocks in the bucket with $E(0)$ blocks. We performed a leaf clearing every eviction - every A accesses.

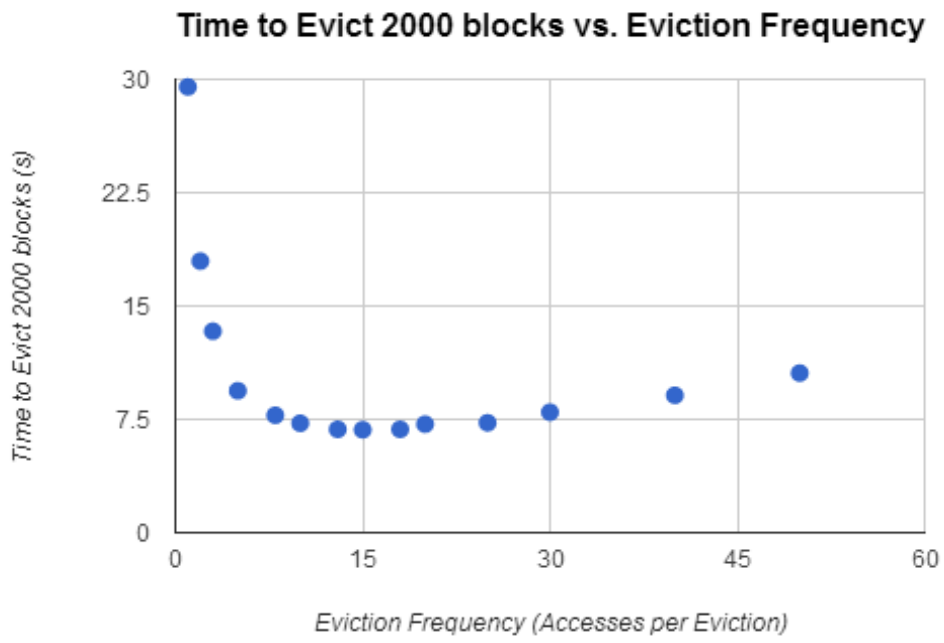

Figure 7: Figure 7: Graph 3: Evaluation of eviction speed for different frequencies

We tested a multitude of eviction frequencies for a tree of 10 levels, with a bucket size $z = 1000$, and found the results in the graph above. Our data suggests an optimal eviction frequency at $A = 15$ for a tree with 10 levels and $z = 1000$. The optimal eviciton frequency is an important consideration for the performance of any C-ORAM, since, as shown above, even a slight change in frequency can result in a major loss of efficiency. Our data also suggests that less frequent evictions are generally more efficient than more frequent evictions (though, of course the right side of the graph has a shallow slope upward, so an excessively infrequent eviction could have other consequences). However, as noted above, evicting too many blocks at a time greatly increases the likelihood of bucket overflow. Although we only conducted a limited test of efficiency, our program can be used as a tool

to determine the optimal eviction frequency for any C-ORAM given the bucket and tree sizes.

4.4 Conclusions and Future Work

In this work, after a study of previous ORAMs, we discuss and analyze C-ORAM, the latest development in constant-bandwidth Oblivious RAM. We suggest various optimizations and also offer a security explanation to improve upon oblivious merging - a key component in C-ORAM. Using a simulation of C-ORAM's eviction process, we obtain data about its performance considering certain parameters such as eviction frequency and bucket size. These contributions are useful in extending the C-ORAM algorithm. In the future, we plan to use our discoveries to further optimize C-ORAM and advance the Oblivious RAM algorithm closer to a usable product for users and businesses.

References

[1] I. Damgård and M. Jurik. A generalisation, a simpli. cation and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography*, pages 119–136. Springer, 2001.

[2] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion oram: A constant bandwidth blowup oblivious ram. Cryptology ePrint Archive, Report 2015/005, 2015.

[3] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM, 1987.

[4] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[5] T. Moataz, T. Mayberry, and E.-O. Blass. Constant communication oram with small blocksize. Cryptology ePrint Archive, Report 2015/570, 2015.

[6] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. Technical report, Cryptology ePrint Archive, Report 2014/997, 2014. http://eprint. iacr. org.

[7] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, 2013.