ORAM paper

Akiva Gordon, Krishna Suraj

Abstract

asdf In this paper, we discuss

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 on 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 on the cloud should be encrypted, such as in online transfers (e.g., PayPal). While current encryption schemes do a good job of protecting actual data, a user's metadata created when accessing and operating on data is still unprotected. For example, if a business were to store data on external or rented servers (e.g., AWS) the untrusted server could still see which files are being accessed, the frequency of accesses, and the time of such accesses. This metadata, or access pattern, is important to protect because otherwise an untrusted server may deduce information about the files themselves. For example, if certain files are accessed more often, the server may deduce that they are important files. Also, if certain files are accessed a lot before a certain data (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 about the user's access pattern of their files.

Even though a server may not know the contents of the files themselves due to encryption, access to metadata can help the server gain information of the relative importance of certain files, etc. Thus, ORAM is important in preserving the online footprints of a user.

In this work, we examine Constant Communication ORAM (C-ORAM) [**?**], 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 reduce the amount of encryption layers needed from Onion ORAM.

2 Previous work

2.1 Previous tree-based ORAM schemes

Initially proposed by Goldreich and Ostrovsky [?], 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, a significant step forward closer the theoretical asymptotic limit for ORAM - on the order of $\log N$. 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. The selected data block is operated on using a read/write operation. All other blocks are placed back on their original paths. The accessed block is assigned a new path 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. If not, a "dummy" block

- a block with meaningless, randomly generated data - is retrieved. The dummy blocks are discarded on the client side.

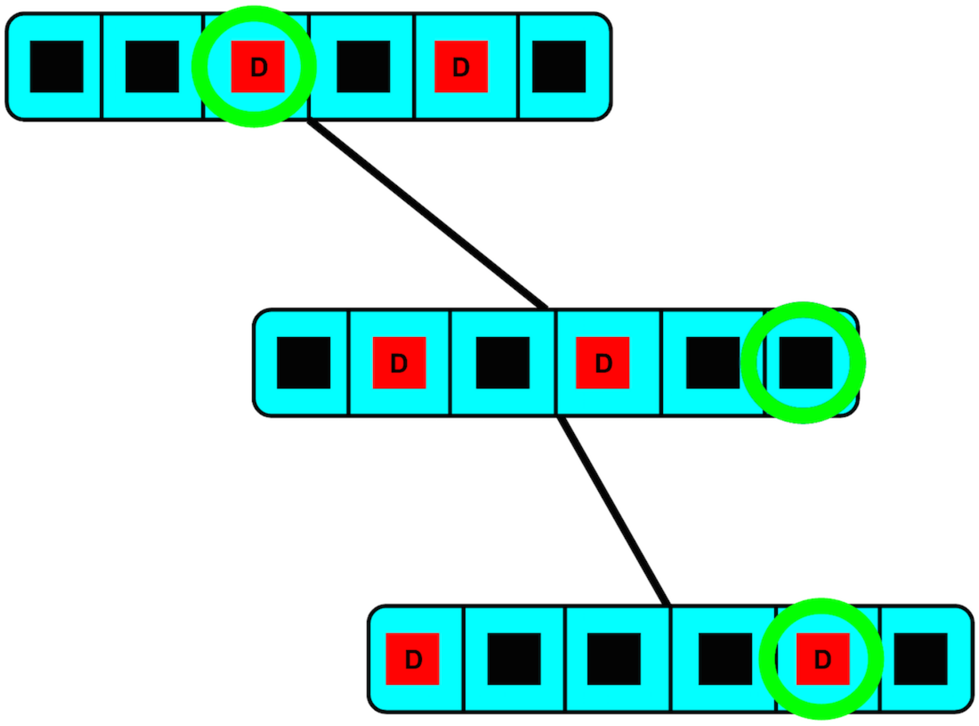

Figure 1: Ring ORAM access. The circled blocks are selected. The buckets which do not contain the target block (levels 1 and 3) send back a dummy block

Figure 1 shows the process of selecting one block from each bucket in Ring ORAM.

There are two changes to the eviction process in Ring ORAM. First, eviction only occurs after every A accesses, in which A is a user-selected parameter. 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.

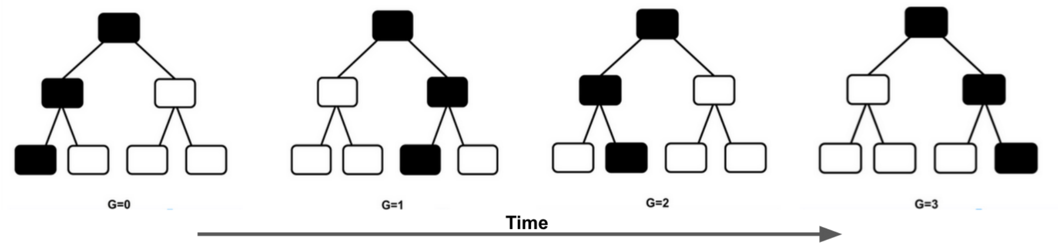

Figure 2: Reverse lexicographical order balances the tree through standardized evictions.

Figure 2 shows the process of selecting eviction paths in Ring ORAM, Onion ORAM, and C-ORAM.

2.2 Onion ORAM

Onion ORAM proposed, using additively homomorphic encryption, the novel idea of having the server conduct some computations for the user rather than all computation being conducted on the user's end. This seems counterintuitive at first - usually, having the server conduct some ORAM functions would reveal essential information to the server about which files are being accessed. However, 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 - a data structure storing the positions and paths of all data. 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. Dummy blocks are indistinguishable from real data blocks to the server, so the access maintains obliviousness. 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 bandwidth. 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 very costly encryption scheme - Damgard Jurik - discussed in the Onion ORAM paper. Encryption in Onion ORAM is the bottleneck, resulting in a poor computational efficiency although a constant bandwidth efficiency successfully breaks the $\log N$ 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

Constant Communication ORAM differs from Onion ORAM in its eviction step. 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. Next, the 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).

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 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. Figure 3 shows the process of oblivious merging.

A main concern with oblivious merging is the security of the permutation conducted by the server to obliviously line up the two buckets. We ensure its security using an intuitive proof. 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.

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.

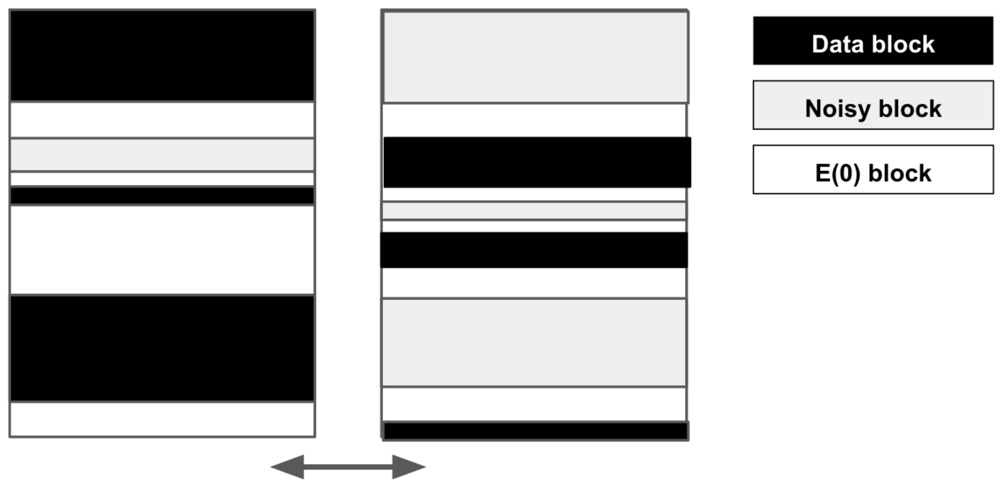

Figure 3: Two buckets are permuted so “real” data blocks (in black) are lined up with either noisy blocks (in grey) or $E(0)$ blocks (in white). Thus, merging the two buckets while preserving real data blocks results in no loss of data.

4 Our Contributions

In our project, we created a simplified C-ORAM protocol (eliminating the encryption for testing purposes). We tested the merging eviction protocol for consistency and loss of data:

4.1 Our Results

We ran tests with ORAMs ranging from 13 to 15 layers with bucket sizes ranging from 500 to 10000.

In our tests, it became clear that the speed of the algorithm is asymptotically dependent on the number of blocks already in the tree. While this does technically seem to be dependent on the constant bucket size, rather than the number of blocks added, it will always be true that the program will be more quickly able to shuffle and re-align buckets when fewer real blocks are present.

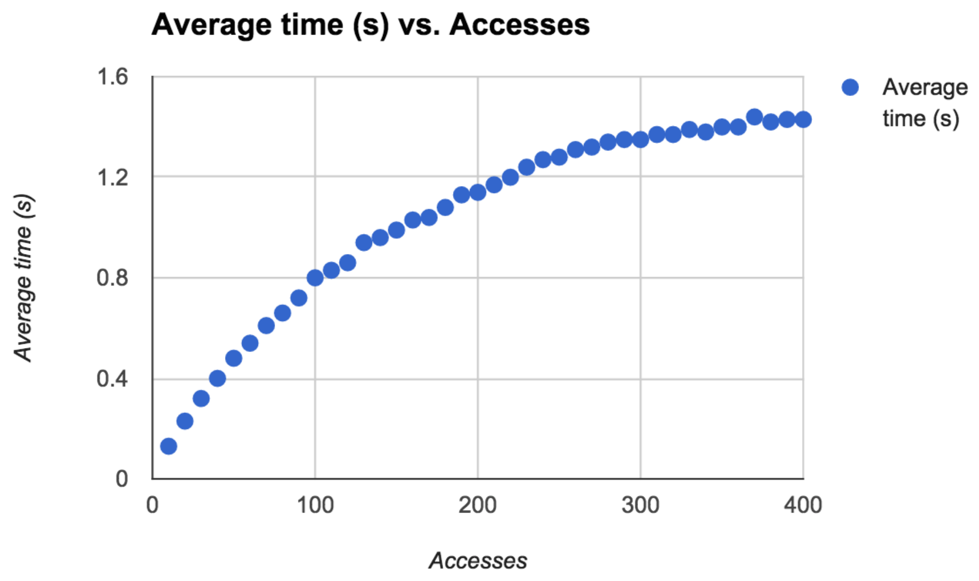

Figure 4:

We were alarmed to learn that after adding z blocks (with no removal from the tree), bucket overflow errors occurred almost every single iteration afterward. This presents a major problem for storing many blocks. This necessitates two operations, first, the removal of blocks from the tree, which must be done at a frequency at least equal to that of eviction. In addition, noisy blocks must be cleaned from the tree as they will quickly accumulate in the leaves if not dealt with otherwise.

In our protocol, we cleaned the leaf of the eviction path to prevent the tree from filling with noisy blocks, but noisy blocks still accumulated quite quickly in higher layers of the tree, frequently resulting in overflows. The only feasible response to this problem is the constant (or near-constant) removal of noisy blocks from the tree, however, this results in a major compromise of security, and invalidates the primary purpose of the noisy blocks to enable higher security without astronomical bandwidth requirements. We propose that, instead of ignoring the noisy blocks or removing all of them, we clean a random number of noisy blocks from each bucket during eviction (this process takes negligible time in comparison to the actual eviction).