ORAM paper

Akiva Gordon, Krishna Suraj, Ling Ren

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. 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 and hacks.

Most data on the cloud is encrypted, such as in important online transfers (e.g. PayPal). While current encryption schemes do an acceptable job of protecting actual data, a user's metadata created when accessing and operating on data is still insecure. For example, if a business were to store data on external or rented servers (e.g. Amazon's 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 (the cloud) 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) [citation here], 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 Terminology

All ORAM schemes discussed in this paper use binary tree-based data structures. A binary tree in computer science consists of a group of nodes, each with two children nodes that it points to. The single node which all nodes are descended from is called the root node. The lowest layer of nodes that have no children are called

leaf nodes. In all modern ORAM schemes, data is organized into blocks. Blocks are the smallest unit of data that can be operated on within an ORAM structure. Blocks come in different varieties. In Onion ORAM, blocks can either contain real data or ”dummy” data, which is randomly generated data that acts as a space-filler. These two types of blocks are indistinguishable to the server because all data is encrypted using Additively Homomorphic Encryption. Blocks are organized further into buckets - which are analogous to nodes on a tree. In all discussed ORAM schemes, one bucket occupies each node of the ORAM tree. ORAM trees exist in the server-space - untrusted and insecure, open to potential snooping. The client accesses, or fetches blocks to the client space - one's own computer. This space is assumed to be fully secure. An eviction is performed occasionally in ORAM to maintain obliquity - i.e. to ”keep the server guessing.” An eviction, at its basic form, consists of a shuffling of blocks on the tree to ensure the server cannot see which blocks are being accessed regularly. Evictions are essential to maintaining security.

3 Previous work

3.1 Pre-Onion ORAM

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.

3.2 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.

3.3 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 - Dmgard Jrik - 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.

4 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.

4.1 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.

4.2 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).

4.3 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.

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.

5 Our Contributions

In our project, we created a simplified C-ORAM protocol (eliminating the encryption for testing purposes).