ORAM PAPER

AKIVA GORDON, KRISHNA SURAJ

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 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. Previous work

2.1. **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.

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.

2.3. **Onion ORAM Eviction.** The eviction step of Onion ORAM consists of all blocks on a certain eviction path being evicted all the way to the leaf nodes of the tree. Each step of this eviction, a new layer of encryption must be added to the data as it is 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.

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. **Overview.** Oblivious merging is a technique that obliviously lines up two buckets in a specific order and merges them into one bucket. Us- ing this technique, we can evict real data elements from a bucket to another by permuting the order of blocks of one of them and then adding additively homomorphically encrypted blocks. Oblivious merging is based on an oblivious permutation generation that takes as input the configurations of two buckets and outputs a permutation ?. A configuration of a bucket specifies which of the blocks in the bucket are real blocks and which are empty. Permutation ? arranges blocks in such a way that there are no real data elements at the same position in the two blocks

3.2. **Construction.** To access an element in C-ORAM, i.e., read or write, the client first fetches the corresponding tag from the position map. This tag defines a unique path starting from the root of the ORAM tree and going to a specific leaf given by the tag. The element might reside in any bucket on this path. To find this element, we make use of a PIR read that will be applied to each bucket. The client downloads the encrypted headers of each bucket. Therewith, the client can generate a PIR read vector retrieving the block from a bucket. To preserve the scheme?s obliviousness, the client sends PIR read vectors for each bucket on the path. Once the block has been retrieved, the client can modify the block?s content if required, then insert it back into the root of the C-ORAM tree using PIR write. This is the standard Path-PIR behavior to read from or write into blocks.

Eviction in our first construction takes place after every ? = O(z) access operations. As in Onion ORAM, a path in C-ORAM is selected following deterministic reverse lexicographic order. Then, the entire root of the ORAM tree is downloaded, randomly shuffled and written back (additively homomorphically) encrypted. Finally, the eviction is performed by repeatedly applying an oblivious merge on buckets along the selected path. Any bucket belonging to this path is obliviously merged with its parent while the other child of the parent will be overwritten by a copy of the parent bucket. We call the former bucket on the path the destination bucket and the latter one its sibling bucket.

Before starting the eviction of a specific path, an invariant of the eviction process is that siblings of buckets of this path are empty, except the leaves. After the eviction, all buckets belonging to the evicted path will be empty except the leaf [6]. Note that siblings of this path, after the eviction, will not be empty anymore. See Fig 2 for a sample eviction with N = 8. Sibling buckets, since they are simply copies of their parents, will contain blocks with tags outside the subtree of this bucket. These blocks are called noisy blocks as they do not belong into this subtree and are essentially leftover ?junk?. Now for correctness, in our construction, we will guarantee that the number of noisy blocks in any bucket is upper bounded. So, there will always be space for real elements in a bucket and will not overflow.

Elements in each bucket are encrypted using additively homomorphic encryption, respectively. Given two buckets B1 and B2, oblivious merging will permute the position of blocks in B1 such that there are no real or noisy element at the same positions in B1 and B2. Consequently, if there is a real element in the ith position in B1, then for the scheme to be correct, the ith position in B2 should be empty. The following addition of elements at the same position in B1 and B2 will preserve the value of the real element. After ? operations, we also download the leaf bucket to delete its noisy blocks.

4. Our Contributions