Oblivious RAM and the Square Root Method

Anurag Kashyap

1 Introduction

In several settings, an eavesdropping adversary can learn sensitive information just from the memory access patterns in a computer/network. An access pattern is defined as a sequence $[A_1, A_2, \ldots]$ of memory addresses that a user/client wants to access in the exact order in which they appear in the list.

Consider the scenario where a client wants to access a large collection of files in the cloud, which are stored in a distributed manner across many computers, and the files are organized by topic in this cluster of computers. The router or Internet Service Provider will be able to see the IP addresses in every request and may be able to deduce sensitive information regarding the program being executed by the client. Analogously, consider the scenario where a malicious program/device has access to the bus connecting the CPU and RAM in a computer, and can see the memory locations being accessed by the CPU. Even in this case, sensitive information about the program being executed on the CPU can be leaked to the malicious third party observing the memory access patterns. The solution to this problem that we discuss in this article is called Oblivious RAM or ORAM. Oblivious here means that the specific sequence of memory locations accessed during the execution of the program are independent of the program. Such obfuscation must necessarily have a price in terms of time and space complexity, but what is that cost? How much slowdown must the program incur to hide the access patterns? Moreover, does such a model of computation even exist?

The seminal paper in this field was published by Ostrovsky where he constructed an Oblivious RAM computer. In this article, we'll see his construction in detail.

2 Construction

The problem is formulated as follows: There is a client/CPU (with access to a PRF, call it F) that wants to obliviously execute a program and access servers/RAM during the course of its execution. The goal is to run the program in a way that the access pattern is independent of the input for any two inputs of the same size.

To see that constructing an ORAM is possible, consider the situation where the CPU scans the whole memory for every instruction in the program. This is clearly an oblivious protocol since the access pattern is independent of the input. However, this protocol is also likely to be very slow and infeasible. The square-root solution is a significant improvement over this model and we will see its construction now.

Suppose we know the amount of memory required by the program in advance, call it N. Henceforth, the term 'virtual memory address' will mean an address in the RAM. The construction can be broken down into 3 parts: Setup, Read Protocol and the Write Protocol.

Setup

In this step, the CPU generates two keys K_1 and K_2. The first key is used to encrypt the data on the RAM and the second key is used to generate the tags for every memory location.

Now, the CPU goes over memory locations 1 to N and augments every memory location in the RAM as follows:

$$RAM_2[i] = < RAM[i], i, tag_i > \tag{1}$$

That is, every memory location now contains its virtual address and a tag along with its original content. The tag_i is the tag for the i-th virtual address, generated as follows $tag_i = F_{K_2}(i)$. The tag will be used to permute the RAM and lookup the contents of the permuted RAM.

Now, we create a new memory RAM_3 which contains all of RAM_2 but also contains $\sqrt{N}$ extra memory locations. For $N+1 \leq j \leq N+\sqrt{N}$, let $RAM_3[j] = < 0, \infty_1 >, tag_j$ where $\infty_1 > N + 2\sqrt{N}$. We call this region of $\sqrt{N}$ memory cells the 'dummy region'.

Since the tags are pseudorandom, if the RAM were to be sorted by the tags, it would be a pseudorandom permutation! This is exactly the next step in the construction, we sort the RAM by the tag values. For any two memory locations i, j memory location i is 'lesser' than location j if $tag_i < tag_j$. In the subsection on Bitonic sort, we will see how this step can be done obliviously.

The next step is to encrypt all of RAM using the secret key K_1 as follows. For $1 \leq i \leq N + \sqrt{N}$

$$RAM_4[i] = Enc_{K_1}(RAM_3[i]) \tag{2}$$

Finally, it appends $\sqrt{N}$ elements to RAM_4, each of which contains an encryption of 0 under key K_1. We call this region the 'shelter' or the 'cache'. Now our setup is complete. The CPU can read any virtual address i by obtaining its tag and then doing an interactive binary search with the RAM to find the location in memory which had virtual address i.

The RAM is reshuffled and encrypted again after every $\sqrt{N}$ I/O operations. A register counter variable keeps track of the number of I/O operations completed.

There are two operations that need to be defined before we explain how read and write protocols work.

The **GET** operation takes as input a virtual memory address i and obtains tag_i and finds the tag in RAM through an interactive binary search. This is possible because the RAM is sorted by tags. In other words, it asks the server for the item stored at location N/2 (assume N is even) decrypts it and compares its tag with tag_i. If tag_i is less than the tag of item ORAM[N/2], then it asks for the item at location N/4; else it asks for the item at location 3N/4; and so on.

Once the item with tag_i has been fetched from ORAM, the CPU decrypts it locally. It then re-encrypts item i (possibly to something different) and asks the memory to store it at the same location.

The **PUT** operation takes as input a virtual memory address and a value *v* to be inserted in that location. It first calls GET(i) as above and then encrypts *v* and stores it at the same location from which GET(i) returned the search result. For the sake of simplicity, we also define another operation **FETCH** that takes a *physical* address in the ORAM as an argument and just returns its contents. We have enough machinery to make one oblivious memory access, but no more. Why? Because if the client were to request the same virtual memory address twice, the RAM would access the same memory location twice too, thereby revealing some information to the attacker. Thus, we must ensure that even if a virtual address is requested twice, the same memory location must not be accessed twice. For this, we will make use of the cache, which will store any contents that have been accessed in the previous $\sqrt{N}$ IO cycles/rounds. In every IO cycle/round, if the command is a read (resp. write) command, then the read (resp. write) protocol is executed.

Read Protocol

Client requests server for virtual address *i* by encrypting it and sending the tag. However, it works a little differently compared to the GET function above. The client first computes tag_i and fetches every location in the cache and checks for equality with tag_i. If the item is found in the cache, we perform a dummy lookup. If not, we call GET[i] and update the cache. The pseudocode below explains how the read protocol works.

```
found = FALSE
z=NONE    //contains the contents of the location we want to read
l=NONE    //contains the virtual address of the location in permuted memory
for j in N+$\sqrt{N}+1 \leq j \leq N+2\sqrt{N}$:
    $temp = DEC_{K_1}(\text{FETCH}[j])$
    If $temp.tag = tag_i$
      $found$ =TRUE
      $z = temp$
      $l = j$
if $found$=TRUE
    GET[$N+round$] (* This is a dummy get *)
if $found$ =FALSE:
    $z$ =GET[i]
    $< contents, addr, tag >= z$
    $val =< contents, \infty_2, tag >$ (* where $\infty_2 > \infty_1$*)
    l = $N+\sqrt{N}+round$ (*where round is 1+number of IO ops done*)
    PUT[$i, val$]
for j in N+$\sqrt{N}+1 \leq j \leq N+2\sqrt{N}$:
    if $j \neq l$ : FETCH[$j$]
```

```
        else: PUT[$j,z$]
    $round$ += 1
```

Write Protocol

The write protocol is similar to the read protocol except for 3 small changes. It also takes a value v along with an address i. If the item is found in the cache, it sets $z = v$ instead of $z = temp$. If the item is not found in the cache i.e., $found$=FALSE, we also execute PUT[i,v] and set $z = v$ before exiting the if-block.

Note that if an item i has already been read before, then a subsequent write to address i will update only the version in the cache and not the one in main memory. The invalid versions in main memory have updated virtual memory addresses of the form ∞_j, which will help to mark and remove them in the restructuring step.

Reshuffling

Whenever a new memory location is accessed for the first time, it is added to the cache. Thus, after $\sqrt{N}$ cycles, the cache could be full and the ORAM will need to restructured. This is done by sorting the entire ORAM based on the virtual memory address stored in each memory location. This performs the task of storing the N memory locations which are either unaccessed or in the cache in the first N locations, since all invalidated and dummy memory locations have their virtual addresses set to ∞_j and so they will occupy the last $2\sqrt{N}$ memory locations. Now the whole memory can be decrypted and encrypted using a new key again, with new tags (also generated using a new key). Now the ORAM is reset and the whole process can be repeated.
The proof that this protocol is oblivious was discussed in class and can be seen in the paper linked in the references. The key idea, however, is that the memory accesses are the same (from the server's point of view) regardless of the input and contents of the cache.

Bitonic sort

To sort the ORAM obliviously, we can use a sorting algorithm called Bitonic sort. Bitonic sort has the property that the memory access pattern remains the same for any input of the same length. This is exactly the obliviousness property! Moreover, since our CPU/client only has constant memory, it can copy two memory locations at a time and compare them, which is sufficient for Bitonic sort. Bitonic sort has a time complexity of $O(n \log^2 n)$.

3 Analysis

Restructuring the ORAM requires calling Bitonic sort which takes $O(N \log^2 N)$ time. Every read or write operation takes $O(\sqrt{N})$ to search the whole cache and $O(\log N)$ for the binary search and $O(\sqrt{N} + \log N) = O(\sqrt{N})$. The initial setup also has a time complexity of $O(N \log^2 N)$. Thus, the amortized cost of the whole protocol is $O(\sqrt{N} \log^2 N)$.

4 References

This article borrows heavily from this **blog post** and Ostrovsky's **original paper**.
Wikipedia link for **Bitonic sort**