ENEE 457

Hiding Your Access Patterns from Google

Prof. Papamanthou
cpap@umd.edu

1 Introduction

In the age of big data, cloud computing comprises the key enabler for processing and analyzing massive amounts of information. Services like Amazon S3 and EC2 offer easily accessible outsourced storage and computation, gradually replacing our local hard drives and desktop machines. Nevertheless, many security concerns have arisen in this new paradigm. In an untrusted cloud setting, users' data and computations can be potentially tampered with and sensitive data could be leaked to unauthorized parties. For cloud computing to be fully adopted by businesses and individuals, security and privacy guarantees must be offered to the clients. For instance, how can one store his private data in the cloud and still enable efficient processing without leaking sensitive information?

In particular, consider the following problem. A cloud server has been set up to store the prices $\mathbf{A}[1], \mathbf{A}[2], \ldots, \mathbf{A}[n]$ of shares $1, 2, \ldots, n$. If someone is interested in buying a share x he would like to somehow download the information $\mathbf{A}[x]$ without letting the server know that he is interested in share x (this information is sensitive since it might reveal the strategy of the client). In this lab we will be coding up algorithms for that task which is called *oblivious access* of an array $\mathbf{A}$.

2 A Simple Algorithm For Oblivious Access: Reading Everything

A simple algorithm to perform an oblivious access is the following. Irrespective of the index x that you would like to access, download all the elements $\mathbf{A}[i]$ from the remote server. After you download the elements, you can easily retrieve $\mathbf{A}[x]$ without the server ever knowing what index you were interested in.

Let n be the size of the array that we want to access obliviously and B be the number of bits that are allocated for each array cell (so that the total size of the array is $n \times B$ bits) Define *oblivious efficiency* $f(n, B)$ of an algorithm as

$$\frac{\text{number of bits that the algorithm downloads to access one element}}{\text{number of bits of that element}}$$

The smaller the above fraction is the better our algorithm is. Years of research work have concluded that this fraction cannot be less than $c \times \log N$ for any constant number c and as N grows very large. What is the oblivious efficiency of the above simple algorithm?

3 A Better Algorithm Using Random Permutations

Consider now an improvement of the above algorithm for the case where a client owns the array $\mathbf{A}$ initially and then uploads it to the cloud so that he can access it obliviously later on. The main idea is the following. First create a random permutation π of the indices $1, 2, \ldots, n$ so that index i is mapped to $\pi[i]$ and so on (the server never gets to see the random permutation—it is kept at the client machine as a secret). Then rearrange the elements of array $\mathbf{A}$ so that element $\mathbf{A}[i]$ is stored at position $\pi[i]$ (and not at position i). Then upload the permuted array at the server. Whenever the client wants to access element i, he downloads $A[\pi[i]]$ and therefore since $\pi[i]$ is a random assignment of i to one element in $\{1, 2, \ldots, n\}$ no information is revealed about i.

What is the oblivious efficiency of the above algorithm? Why does it contradict the $c \times \log N$ lower bound that we talked about before? Does this algorithm really provide an oblivious access? What information is revealed to the server if the client wants to access index 10 one million times?

4 An Improved Algorithm with Sublinear Oblivious Efficiency

Consider now a different algorithm:

Setup: Given an array $\mathbf{A}$ of n elements, the client extends $\mathbf{A}$ into an array of $n + \sqrt{n}$ elements, with the last $\sqrt{n}$ elements being the dummy elements. Then the preparation algorithm permutes $\mathbf{A}$ into $Aperm$ of $n + \sqrt{n}$

elements such that $\mathbf{A}[i]$ is stored at $Aperm[\pi[i]]$. The algorithm also initializes an array $\mathbf{C}$ of $\sqrt{n}$ elements that initially stores dummy elements. The client uploads the permuted array $Aperm$ and $\mathbf{C}$ to the server and stores the permutation π locally.

Oblivious access: To access obliviously index x the algorithm works as follows. Set $count = 1$ to be a variable counting the number of accesses.

1. First, the client downloads array $\mathbf{C}$. If $\mathbf{A}[x]$ is not stored in $\mathbf{C}$, then the algorithm accesses $Aperm[\pi[x]]$ to retrieve $\mathbf{A}[x]$. If $\mathbf{A}[x]$ is in $\mathbf{C}$ then the client has found $\mathbf{A}[x]$ and for security purposes he accesses the dummy position $\mathbf{A}[\pi[n + count]]$;
2. Store $(x, \mathbf{A}[x])$ in the first available position of array $\mathbf{C}$ (the position *count*) and upload array $\mathbf{C}$ again. Increment *count*.
3. Finally, if $count > \sqrt{n}$, download *Aperm*, reshuffle it using a new random permutation π', keep π' locally and upload *Aperm* again. Set $count = 1$.

Answer the following questions:

1. Why do you think the above algorithm is secure? Argue how the above algorithm avoids the problems of the previous algorithm.
2. What is the worst value of oblivious efficiency of the algorithm?
3. What is the average value of oblivious efficiency of the algorithm?