Computer Science Project Proposal

Encrypted Keyword Search Using Path ORAM on MirageOS

R. Horlick, Homerton College

Originator: Dr N. Sultana

23 October 2015

Project Supervisors: Dr N. Sultana & Dr R. M. Mortier

Director of Studies: Dr B. Roman

Project Overseers: Dr M. G. Kuhn & Prof P. M. Sewell

Introduction and Description of the Work

As the cost of large-scale cloud storage decreases and the rate of data production grows, more and more sensitive data is being stored in the cloud. We, of course, want to encrypt our data, to ward off prying eyes, but this comes at a cost. We can no longer selectively retrieve parts of the data at will. We need some method of searching over encrypted data to find the parts we are interested in.

So let us say that Alice has a set of documents that she wants to store on an untrusted server, run by Bob. We'll first assume that Bob is "honest, but curious", that is, he will attempt to gather all knowledge that he can without deviating from the protocol. Alice wants to store her documents encrypted, but also wants to search over them without Bob being able to learn either the keywords she is searching for, or the results of any query, the documents that contain the keyword.

There are a number of schemes in the literature that use symmetric encryption techniques to build a searchable encryption scheme. The problem is that these all leak the access pattern, so Bob knows which documents matched any query, even if he doesn't know what they matched. It turns out that this pattern of access can leak large amounts of information. In a study [1] on an encrypted email repository, up to 80% of search queries could be inferred from the access pattern alone! So clearly this a leak worth plugging, but how can we do it?

One solution to our problem is to use Oblivious Random Access Memory (ORAM), a cryptographic primitive that hides data access patterns. That is, we turn Bob's server into a block device and we attempt to maintain the property that any two sequences of accesses of the form $(operation, address, data)$, that are the same length, have computationally indistinguishable physical access patterns. Bob should have no way of learning what $address$ we are really accessing.

A trivial ORAM algorithm operates by scanning over the whole ORAM and reading/updating only the relevant block, but this has $O(N)$ bandwidth cost, where N is the number of blocks, which is highly impractical for large-scale storage. Luckily, much better algorithms have been proposed. We choose to focus on Path ORAM, because it has only $O(\log N)$ bandwidth cost in the worst case if $B = \Omega(\log^2 N)$, as well as being incredibly simple conceptually.

Now let's assume that Bob has become malicious, and is modifying our encrypted data. In order to combat this, we can provide integrity verification by treating the ORAM as a Merkle tree, but with data in every node. The details of this scheme are outlined below after Path ORAM has been described further.

So the project is a searchable encrypted object store, with integrity verification. It will provide a simple, name-value pair API, that allows more complex filesystems to be built on top of it. A block diagram of the system is shown in Figure 1.

Starting Point

ORAM protocols generally operate in a client-server model. For our purposes the server is the cloud storage provider and the client will be a MirageOS cloud instance. MirageOS is a unikernel operating system designed to run on the Xen hypervisor. I have chosen to use Mirage for a number of reasons. Firstly, it is lightweight and designed to be run in the cloud, meaning that simple cloud services can be built on top of it that fully leverage the ORAM. Secondly, it is written in OCaml, meaning that I can take full advantage

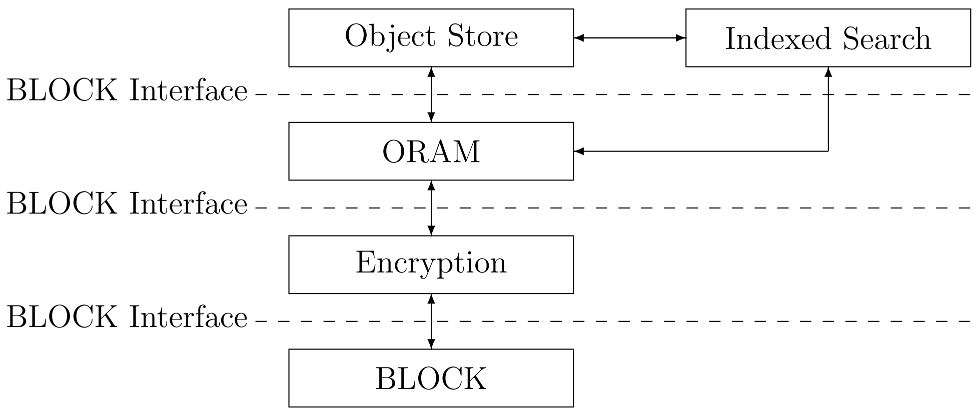

Figure 1: The Application Stack: We can use any underlying BLOCK implementation and we can add/remove ORAM, Encryption or Search modules as we please

of the rich module system. This will allow me to write my implementation as a functor that takes an implementation of Mirage's BLOCK interface and creates a new BLOCK implementation that uses ORAM, as shown in Figure 1. This means that the ORAM could be used with any underlying implementation of the BLOCK interface and that it would plug seamlessly into any existing program.

MirageOS is a framework, that pulls together a number of libraries and syntax extensions, to provide a lightweight unikernel operating system. It provides a command line tool for generating the main file, that links together implementations of various parts of the system, and passes them to the unikernel. There are a number of module signatures that define the operation of devices, such as CONSOLE for consoles, ETHIF for ethernet, and most importantly for us BLOCK, for block devices.

There are currently two implementations of the BLOCK interface, one for Unix and one for Xen, and I would like to support both. Thus I will build a functor that takes one of these and emulates a block device using it, adding ORAM to it. This will most likely use some of the algorithms/code from the existing implementations, for dealing with buffers, implemented using the Cstruct library.

Substance and Structure of the Project

Substance

The Path ORAM protocol has three main components: a binary tree, a stash and a position map. The binary tree is the main storage space. Every node in the tree is a bucket, which can contain up to Z blocks. The tree has height L, where the tree of height 0 consists only of the root node, and the leaves are at level L. The stash is temporary client-side storage, consisting only of a set of blocks waiting to be put back into the tree. The position map associates, with each block ID, an integer between 0 and $2^L - 1$. The invariant that the Path ORAM algorithm maintains is that if the position of a block x is p, then x is either in some bucket along the path from the root node to the p^{th} leaf, or in the stash. On every access to the tree, a whole path is read into, the accessed block is assigned a new random position and then as many blocks as possible are written back into the same path. This means that in any two access to the same block, the paths that are read are statistically independent.

We can extend the basic path ORAM algorithm with recursion. That is, calling the data ORAM $ORAM_0$, we store the position map of $ORAM_0$ in a smaller ORAM, $ORAM_1$, and the position map for this in an even smaller ORAM, $ORAM_2$. We can do this until we have a sufficiently small position map on the client. Supposing that we store χ leaf addresses in each PosMap ORAM, the position for a data block with address a_0 is at $a_1 = a_0/\chi$ in $ORAM_1$, and in general $a_n = a_0/\chi^n$ for the address in $ORAM_n$.

Recursion does add an overhead, but we can reduce this overhead by exploiting locality. Assuming that programs will access adjacent data blocks, we can cache PosMap blocks in a PosMap Lookaside Buffer, so that if all χ data blocks that are referenced in a PosMap block are accessed in turn we only need to do the recursion once. Doing this naïvely, however, breaks security, because we are revealing information through the cache hit pattern. To avoid this we use Unified ORAM, which combines all of the recursive ORAMs into a single logical tree. We then use the address space to separate the levels of recursion, so addresses 1 to N are for data blocks, $N+1$ to $N+(N/\chi)$ for $ORAM_1$ and so on. Now all accesses occur in the same tree, and the security of Path ORAM keeps the cache miss pattern hidden.

A final optimisation compresses the PosMaps, reducing the number of levels of recursion required to achieve the desired client side storage, resulting in an asymptotic bandwidth complexity for ORAM with small block size.

The last two optimisations were originally designed and tested in a secure processor setting, so their application to the cloud storage setting is novel. The evaluation section of this project will provide empirical evidence as to whether these optimisations are worthwhile for our purposes.

Structure

The project breaks down into the following sub-projects:

1. Familiarising myself with OCaml, MirageOS and related libraries
2. Implementing the basic Path ORAM functor and testing that it works in place of existing BLOCK device implementations
3. Implementing the three main optimisations to Path ORAM as extensions to the same functor, allowing for the use of different combinations of optimisations
4. Creation of a suite of tests and experiments to evaluate the relative performance of the optimisations and their conformity to theoretical bounds
5. Writing the dissertation

Success Criterion for the Main Result

To demonstrate, through well chosen examples, that I have implemented a secure Path ORAM functor, along with a number of "optimisations".

Possible Extensions

Path ORAM (and other tree based ORAMs) are limited in the fact that they have fixed-sized trees. Thus, we either need to know our storage requirements before setting up the ORAM, potentially wasting resources, or resize them in the naïve way as storage requirements increase. If I achieve the goals of my main project, including evaluation, ahead of schedule I will examine the possibility of making Path ORAM resizable.

Timetable: Work plan and Milestones

Planned starting date is 16/10/2015.

1. **16/10/15 – 26/10/15** Familiarise myself with relevant Mirage libraries. Implement basic Path ORAM functor.
2. **27/10/15 – 09/11/15** Implement basic test harness. Start implementation of recursive Path ORAM.
3. **10/11/15 – 23/11/15**
4. **24/11/15 – 04/12/15**
5. **05/12/15 – 18/12/15**
6. **18/12/15 – 31/12/15**
7. **01/01/16 – 08/01/16**
8. **09/01/16 – 29/01/16**
9. **30/01/16 – 08/02/16**
10. **09/02/16 – 21/02/16**
11. **22/02/16 – 06/03/16**
12. **07/03/16 – 11/03/16**
13. **12/03/16 – 25/03/16** Final draft
14. **01/05/16 – 13/05/16** Reread and make any final edits and then submit

Resources Required

- My own laptop for implementation and testing
- My own external hard disk for backups
- GitHub for version control and backup storage
- MirageOS libraries as a basis for the project