Rupert Horlick

Encrypted Keyword Search Using Path ORAM on MirageOS

Computer Science Tripos – Part II

Homerton College

January 12, 2016

Proforma

Name:	**Rupert Horlick**
College:	**Homerton College**
Project Title:	**Encrypted Keyword Search Using Path ORAM on MirageOS**
Examination:	**Computer Science Tripos – Part II, July 2016**
Word Count:	**1587**[1] **(well less than the 12000 limit)**
Project Originator:	Dr Nik Sultana
Supervisors:	Dr Nik Sultana & Dr Richard Mortier

Original Aims of the Project

Work Completed

Special Difficulties

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Rupert Horlick of Homerton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Todo list

Chapter 1

Introduction

1.1 Motivation

Cloud computing is becoming ubiquitous, with more than an exabyte of data estimated to be stored in the cloud. For large businesses, a private cloud can be a cost effective way to keep data isolated, but as public cloud services become ever cheaper, even these businesses could be forced to succumb to market pressures and move to the public cloud. With so much important data held by only a few major cloud providers, trust becomes a major issue.

Encryption appears to be the solution to our trust issues; if the providers cannot read the plain-text of our data then surely it is secure? This appears to hold in general, but in the important application of query-based searching, we have a problem: using current methods of homomorphic encryption to perform search over encrypted documents can leak up to 80% of queries [2]. Knowledge about the queries made to a data set, along with the amount of documents returned by each query, could lead to some dangerous inferences. As a motivating example, discovering that a query such as $\langle name,\ disease \rangle$ made to a medical database returned results would allow an adversary to uncover information about a patients medical status, breaching patient confidentiality.

What allowed the authors of [2] to infer search queries was knowledge about the documents returned by any specific query. Thus, in order to protect against this kind of attack we need to have some way of preventing the server from knowing which documents it is actually returning in response to any query. Oblivious Random-Access Memory (ORAM) gives us exactly what we want. When using ORAM, not only are two accesses made to exactly the same piece of data computationally indistinguishable to the server, but so too are any two access patterns of the same length.

This project aims to demonstrate that, using a particular ORAM protocol Path ORAM [6], it is possible to build a system that allows us to search over a set of encrypted documents without leaking the resulting access pattern, protecting the content of the search queries and therefore the confidentiality of the documents.

1.2 Challenges

The first major challenge that faces any security related project is adequately defining the threat model. In order to be able to reason about and evaluate the security properties of the system, we need to know exactly what we assume an adversary to be capable of. Once we have done this we must show that within these capabilities, the security properties that we desire the system to have remain intact. The threat model will be defined in section 2.1.

By virtue of being stored in the cloud, we should be able to access our data at any time, from any place, while still maintaining the desired security properties. We also want to be tolerant of network connection errors and client-side crashes. Thus, another challenge is to make the system completely stateless.

In order to make statelessness more efficient, it is necessary to augment ORAM with recursion (section 2.2), which reduces the amount of transient client-side storage. This enables us to efficiently store the client's state between accesses. Recursion introduces the challenge of choosing how many levels to use. Each extra level of recursion reduces the size of the client's state, but also incurs a time and space overhead. This is explored briefly in [6], but we will attempt to have the system automatically choose parameters for the recursion based on the size and block size of the underlying storage used by the system.

Finally in order for the system to perform query-based search over encrypted documents we need to implement an object store, an information retrieval module and an encryption module. These modules will be rudimentary, as the main focus of the project is the implementation and optimisation of the ORAM module, but all of them have inherent design and implementation challenges that will be need to be addressed.

1.3 Related Work

Explore and discuss related work

Chapter 2

Preparation

2.1 Defining the Threat Model

The threat model is defined in terms of a client, a server and an attacker.

We assume that the network is under the attacker's control. In the basic model we define the server and the attacker to be honest, but curious. This means that they will both gather as much information as possible, without deviating from the protocol. Thus the attacker will eavesdrop, but will not prevent messages from arriving at the server, tamper with them in any way or produce their own fake messages. The server will also not tamper with the messages, or with the underlying storage that the protocol is accessing. In this model the attacker is not as interesting as the server, because all communications are encrypted. Thus it is the server, that can see the access patterns of the underlying storage, that we are trying to protect against. The goal of the project is to make sure that this access pattern reveals nothing about the contents of the search queries or the documents in the underlying storage.

We can also extend the model to one where both the attacker and the server freely tamper with messages and introduce new messages, and the server tampers with the underlying storage. In this case we will need to extend our solution with integrity verification.

2.2 Introduction to Path ORAM

Path ORAM is defined in terms of a client and a server, where the client wishes to store data on the server. The data is split into blocks and each block is tagged with a sequence number or address, its position if the data were to be stored sequentially. On the server the data is stored in a binary tree of height L, where each node in the tree is a bucket with size Z, containing up to Z data blocks. The client requires two local data structures. The stash is a sort of working memory; as data is read from the server it is put into the stash and it is then written back to the server later on. The position map associates with each address a number between 0 and $2^L - 1$, which corresponds to a leaf in the tree. The protocol maintains the invariant that, at anytime, a block with position x in the position

map is either in the stash, or in a bucket along the path from the root of the tree to the x^{th} leaf.

This is achieved using algorithm 1, which is split into four main steps:

Remap Block The current position of the block a is read and then a new position is calculated uniformly at random

Read Path The path to the leaf x is read into the stash. At this point the block for address a is definitely in the stash, if it has ever been written into the ORAM

Write New Data If the operation is a write, then the value in the stash is replaced by the new data^*

Write Path The path to the leaf x is filled with blocks from the stash. A block with address a' can be put in the bucket at level l, if the path to $\mathsf{position}[\mathsf{a}']$ intercepts the path to x at level l. If $min(|S'|, Z) < Z$, then the bucket is filled with dummy blocks

Clearly after this, either it was possible to write a block into the stash along the path to its leaf, or not, in which case it is in the stash, and the invariant holds.

Algorithm 1 Read/write data block at address a

function ACCESS($\mathsf{op}, \mathsf{a}, \mathsf{data}^*$)
  $x \leftarrow \mathsf{position}[\mathsf{a}]$
  $\mathsf{position}[\mathsf{a}] \leftarrow$ UNIFORMRANDOM($2^L - 1$)
  for $l \in \{0, 1, \ldots, L\}$ **do**
    $S \leftarrow S \cup$ READBUCKET($\mathcal{P}(x, l)$)
  end for
  $\mathsf{data} \leftarrow$ Read block a from S
  if $\mathsf{op} = \mathsf{write}$ **then**
    $S \leftarrow (S - \{(\mathsf{a}, \mathsf{data})\}) \cup \{(\mathsf{a}, \mathsf{data}^*)\}$
  end if
  for $l \in \{L, L-1, \ldots, 0\}$ **do**
    $S' \leftarrow \{(\mathsf{a}', \mathsf{data}') \in S : \mathcal{P}(x, l) = \mathcal{P}(\mathsf{position}[\mathsf{a}'], l)\}$
    $S' \leftarrow$ Select min($|S'|, Z$) blocks from S'
    $S \leftarrow S - S'$
    WRITEBUCKET($\mathcal{P}(x, l), S'$)
  end for
  return data
end function

The security of this operation comes from the fact that positions are assigned uniformly at random. If we consider the sequence of accesses,

$$\mathbf{p} = (\mathsf{position}_M[\mathsf{a}_M], \mathsf{position}_{M-1}[\mathsf{a}_{M-1}], \ldots, \mathsf{position}_1[\mathsf{a}_1]),$$

any two accesses to the same address will be statistically independent and trivially so will two accesses to different addresses. Thus, by an application of Bayes' rule,

$$\Pr(\mathbf{p}) = \prod_{j=1}^{M} \Pr(\mathsf{position}_j[\mathsf{a}_j]) = \left(\frac{1}{2^L}\right)^M$$

and the access pattern is indistinguishable from a random sequence of bit strings.

Give a quick introduction to recursive oram and statelessness

2.3 Introduction to Inverted Indexes

The inverted index is the single most important data structure in Information Retrieval. It allows us to perform a large amount of work in advance in order to give performance that is linear in the number of documents involved in a query, much better than a linear scan of all documents that is $O(NK)$, for N documents with an average of K terms. It consists of two parts: the dictionary and the postings. The dictionary is a list, usually stored as a hash table, of all of the terms that appear in a set of documents. Then for each term, we have a postings list, a list of all the documents that contain a term. Collectively these postings lists are referred to as the postings.

So now looking up a single keyword is as simple as hashing the keyword and returning the relevant postings list, if it exists. Simple boolean operations are also easy to support. Disjunction simply takes the union of two postings lists and conjunction the intersection.

In this project we will restrict operations to simple, space-separated disjunctions, because we are focusing on showing the correctness and efficiency of search using the ORAM implementation, rather than creating an advanced IR system.

Talk about constructing an inverted index

2.4 Introduction to MirageOS

Give a brief introduction to MirageOS

Discuss the choice to use MirageOS

2.5 System Architecture

Give an overview of the system level design including the figure from the proposal

2.6 Requirements Analysis

High Priority Basic ORAM implementation

Medium Priority Object Store

Medium Priority Search Module

Low Priority Encryption

Low Priority Further Optimisations including Recursion and Statelessness

Low Priority Integrity Verification (Extension)

Further discuss the requirements identified above

2.7 Choice of Tools

2.7.1 OCaml

Having decided to build ORAM on top of MirageOS, OCaml was really the only choice of programming language, because Mirage applications and libraries are all built in OCaml. However, OCaml was also part of the reason for choosing to build on Mirage.

OCaml has an extremely powerful module system, making it easy to parameterise the ORAM implementation over module signatures for Block devices or other system components. This not only encourages more generalised programming, but also allows powerful techniques like recursive modules to be exploited. For instance, to implement recursive ORAM, the position map of the main ORAM is another ORAM, so we can simply parameterise over a position map signature and pass the ORAM implementation to itself as the position map.

OCaml's static typing system is also indispensable for ensuring correctness of programs and increasing productivity.

2.7.2 Libraries

Add a small amount of discussion to this section

2.7.3 Development Environment

Add a small amount of discussion to this section

2.8 Software Engineering Techniques

I employed two major techniques to ensure my code was well thought through and well built, while remaining productive.

Library	*Version*	*Purpose*	*License*
Mirage	2.6.1	System Component Interface Definitions, Application Configuration Framework	ISC
Jane Street's Core	112.35.00	Data Structures, Algorithms	Apache-2.0
LWT	2.5.1	Threading	LGPL-2.1
Cstruct	1.7.1	Data Structure	ISC
Alcotest	0.4.6	Unit Testing	ISC

Table 2.1: Libraries used by Mirage ORAM

Tool	*Version*	*Purpose*	*License*
Mac OSX	10.11.2	Operating System	Proprietary
Atom	1.3.2	Text Editor	MIT
OPAM	1.2.2	Package Manager	GPLv3
OASIS	0.4.5	Build Tool	LGPL-2.1

Table 2.2: Tools used in the development of Mirage ORAM

First of all a Test Driven Development approach allowed me to fail fast. I wrote unit tests for each new piece of code that was written, until I was sure that it was working correctly. This meant that when it came to combining small modules into a larger system, things worked together as expected more often than not.

Secondly, I wrote interface files before writing the actual implementation. This meant that I had to think about the design of each module thoroughly before writing any code and also meant that I could make adjustments to other modules in order to fit with the new design.

Combining these techniques with documentation and structuring of both the source code and the source repository, led to a manageable and feasible development workflow.

2.9 Summary

In this chapter I have discussed the work that was undertaken before development began. This included a rigorous definition of the threat model, a brief introduction to the major algorithms, data structures and libraries, an overview of preliminary architectural designs, and a discussion of the techniques and tools used in the development process.

The next chapter will show how all of this information was used to achieve the aims of the project.

Chapter 3

Implementation

This chapter describes the process that took the designs and algorithms of the previous chapter and turned them into a functioning system. As it is the core of the project, the implementation of Path ORAM is discussed first (section 3.1), followed by the object store (section 3.2), the search module (section 3.3) and finally encryption (section 3.4).

3.1 Path ORAM

The structure of Path ORAM is described abstractly in section 2.2, in terms of the core data structures and the access algorithm. In this section we discuss how those data structures were realised and the design decisions involved in the implementation.

3.1.1 Inherent Constraints

By virtue of writing an implementation to satisfy an existing module signature, there are a number of constraints put on the design of our system.

The first major constraint is the use of the Cstruct library, discussed in section 2.7.2. The underlying block device requires a buffer of type `Cstruct.t` to read to/write from and in order to satisfy the `BLOCK` module signature, ORAM needs to return data as a `Cstruct.t`. Thus, to avoid unnecessary conversions, we will pass our data around in this form.

Another constraint is the usage of `int64` as the type of addresses in `BLOCK`'s, `read` and `write` operations. Again, to avoid unnecessary (and potentially unsafe) work converting between types, we will use `int64`s wherever necessary.

3.1.2 Stash

The stash stores blocks of data temporarily before they are written back into ORAM. It needs to support operations of insertion, lookup based on address and removal. For this job I chose an `int64`-keyed hash table from Jane Street's Core library, with `Cstruct.t` values. This was put into its own module, abstracting away the underlying type, meaning that we could swap implementations of the Stash module without breaking the core code of the ORAM module.

Security Parameter (λ)	Bucket Size (Z)		
	4	5	6
	Max Stash Size		
80	89	63	53
128	147	105	89
256	303	218	186

Table 3.1: Empirical results for maximum persistent stash size

The hash table gives us constant time for the operations of insertion, lookup and removal, making it ideal for our needs. The hash table implementation takes an initial size as a parameter, and expands when necessary. This would add a large overhead, because we would have to copy the entire contents of the stash. However, as shown in [6], we require exactly $Z \log_2 N$ transient storage, where N is the size of the ORAM in blocks, and on top of that, we require a constant amount of space for persistent stash storage. Table 3.1 shows the maximum stash size required depending on the security parameter, λ, and the bucket size Z. A stash with security parameter λ has probability $2^{-\lambda}$ of exceeding this stash size.

In order to achieve statelessness, we need to be able to write the stash to disk, so there is going to be a trade-off between maximum stash size and security parameter. A larger maximum stash size increases bandwidth, but a security parameter set too low will not allow us to allocate the storage space for the stash in advance. For the purposes of this project, we will parameterise in both bucket size and security parameter, allowing us to discover empirically the values that optimise the construction.

3.1.3 Position Map

The position map associates a leaf position with each address of the data. As mentioned in section 3.1.1, we are constrained to using `int64` addresses, thus we need to be able to support a position map that is indexed by 64-bit integers. OCaml provides us with a Bigarray module, but the size of these arrays is specified using the OCaml `int` type. This type actually uses only 63 bits on a 64-bit machine and 31 on a 32-bit machine. Both of these types are also signed, so we need to represent a type that can range up to $2^{63} - 1$, using a type that can only go up to $2^{30} - 1$ on a 32-bit machine.

In order to do this we need to use 3-dimensional arrays. We take an `int64` value and split its bits into a 4-bit value and 2 30-bit values. The 4-bit value is the most significant bits, and will therefore be 0 unless we store more than 2^{60} blocks. The 30-bit values are guaranteed to be converted into positive `int`s, which we can then use to address two dimensions of the array. We have to perform some input sanitation to make sure that the 4-bit value (and therefore the 64-bit value) is positive.

Now the only remaining problem is creating the position map. Algorithm 2 shows how we translate from a desired `int64` size to the dimensions of a three dimensional array. After splitting the `int64` as described above, we must add one to the first two dimensions, because we always want them to be at least of size 1. If a higher dimension is greater

than 1, then all lower dimensions become their maximum value, in this case $2^{30} - 1$. We can now create an array using these values that is guaranteed to be at least the size that we require on both 32-bit and 64-bit machines.

Algorithm 2 Calculate the dimensions of a 3D array given total desired size

Require: size > 0

```
1:  function PosMapDims(size)
2:      (x, y, z) ← SplitIndices(size)
3:      x ← x + 1
4:      y ← y + 1
5:      if x > 1 then
6:          y ← 0x3FFFFFFF
7:          z ← 0x3FFFFFFF
8:      else if y > 1 then
9:          z ← 0x3FFFFFFF
10:     end if
11:     return (x, y, z)
12: end function
```

3.1.4 Creating ORAM

We want ORAM to be able to replace any existing block device in any Mirage program. In order to do this we need access to the methods of the underlying block device as well as the block device itself. Thus, we need to build an ORAM functor, that takes a module satisfying the `BLOCK` interface, shown in listing 2, and gives us a new module satisfying the same interface.

In order to get access to this underlying device, we need a `create` method, which takes a block device as input and returns something of type `Oram.Make(B).t`, shown in listing 1. This type contains the ORAM parameters such as `bucketSize` and `offset`, structural information such as the `height` of the ORAM and the `numLeaves`, and pointers to the stash, position map, and underlying block device.

The following parameters are passed as input to the `create` method, along with the block device:

`size` The desired size of the ORAM in blocks

`blockSize` The desired size of a single block in bytes

`bucketSize` The number of blocks in a bucket

Using these, the `create` method can calculate new structural information. First we need to calculate the number of sectors required for a block as

$$\texttt{sectorsPerBlock} = \frac{\texttt{blockSize} - 1}{\texttt{sector\_size}} + 1,$$

which rounds up the number of sectors so we can always fit the desired block size. Next, we calculate the sector size of the ORAM as

$$\texttt{sector\_size} = \texttt{sector\_size} \times \texttt{sectorsPerBlock} - 8,$$

which is the block size, minus 8 bytes for the address, which needs to be stored along with the data. Now we can calculate the height of the ORAM, but we need to consider two cases. If the desired size of the ORAM is specified, then we simply calculate the height as

$$\texttt{height} = \left\lfloor \log_2 \left(\frac{\texttt{size}}{\texttt{bucketSize}} + 1 \right) \right\rfloor - 1.$$

This comes from rearranging the equation for the size of the binary tree, $2^{\texttt{height}+1} - 1$, introducing the floor operator so that the resulting binary tree is less than or equal to the desired size. If the size is unspecified, we assume that we should fill as much of the device as possible, so we calculate

$$\texttt{size} = \frac{\texttt{size\_sectors}}{\texttt{sectorsPerBlock}}$$

and then perform the same calculation as above with this new value. Finally we calculate, `numLeaves` and a new value for `size_sectors` trivially.

This is all of the structural information we require, so now we only have a couple of things left to do. We must create instances of the client-side data structures and we must initialise the ORAM space. The first simply calls the creation functions of the associated data structures. The second loops through the block device, writing dummy blocks to every location. Dummy blocks have address -1, and are ignored by the access protocol. Now we can return everything that we have as an instance of `ORAM.Make(B).t`.

Listing 1 The type of an ORAM device `ORAM.Make(B).t`

```
type parameters = PositionMap.parameters

type structuralInfo = {
  height : int;
  numLeaves : int64;
  sectorsPerBlock : int;
}

type t = {
  info : info;
  structuralInfo : structuralInfo;
  parameters : parameters;
```

3.1.5 Accessing ORAM

Now that we have the client-side data structures and our ORAM module, we can almost implement algorithm 1. Before we can though, we need the surrounding plumbing. The `BLOCK` interface functions `read` and `write` input/output data as a list of `Cstruct.t`s, so they must split this into chunks before calling `access` on each one. On the other side of access, we need the subroutines ReadBucket and WriteBucket, which access the underlying block storage.

It is the second two of these plumbing functions that have a much more interesting role. They are actually responsible for maintaining the logical binary tree structure. There are no physical pointers, but instead the structure is built purely through calculating the appropriate address of a bucket. The bucket on the path to leaf x at level l is calculated using algorithm 3.

Algorithm 3 Calculating the address of the bucket at level l on the path to leaf x

```
function BucketAddress(x,l)
    address ← 0
    for i = 0; i < l; i++ do
        if x >> (i + height − l) && 1 = 1 then
            address ← (2 × address) + (bucketSize × sectorsPerBlock × 2)
        else
            address ← (2 × address) + (bucketSize × sectorsPerBlock)
        end if
    end for
    return address
end function
```

This is most easily explained using fig. 3.1. Here we can see the nodes labelled in order of their position in memory. The leaves are also labelled, but the binary representations of these labels are actually more important. The binary representation, read from left to right, tells us how to reach the leaf. To get to leaf 5, with binary representation 101, we go right, left, right. When we go left, we double the label on the node and add one, and when we go right, we add two. Multiplying this node label by the block size and the bucket size gives us the physical address of the node. So for a node at level l, we only compare the first l bits, following the procedure above.

So we can now input/output data from the outside and the underlying block device. This leaves most of algorithm 1 fairly trivial to implement. Remapping the blocks requires a simple call to a pseudo-random function, reading in the path just adds the contents of each bucket to the stash, and writing new data is as simple as setting a value in the stash. The only interesting part left is lines 11-16. This is where we decide which blocks should be put into the path.

The naive implementation of this, and that suggested by algorithm 1, loops through the stash and finds blocks such that the bucket at level l on the path to leaf $\mathsf{position}[\mathsf{a}']$ is the same as the bucket at level l on the path to leaf x. There are two small optimisations

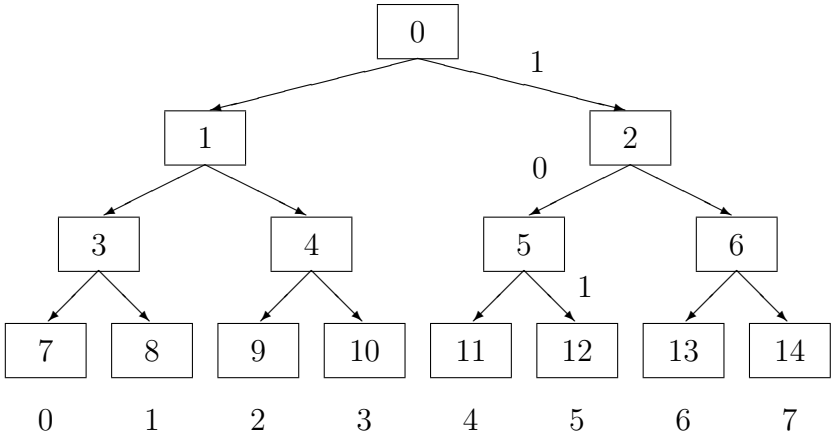

Figure 3.1: Visualisation of algorithm 3

we can make here. The first is to perform the position lookup only once, attaching positions to the blocks in a temporary data structure, avoiding repeated work at each level. The second avoids calculating the bucket addresses entirely. We can do this by noting that in order for the paths to two leaves to intersect at level l, the leaves must have the same first l bits. Thus, we can simply perform a right bit shift on both x and $\mathsf{position}[\mathsf{a}']$ of $\texttt{height} - l$ bits, and check for equality.

So at this point we have a functioning ORAM functor, that can be used to augment a block device in any Mirage program. In sections 3.1.6 to 3.1.8 we look at optimisations and extensions to this ORAM construction, before moving onto to the further aims of the project.

3.1.6 Recursion

THIS SHOULD PROBABLY BE MOVED TO CHAPTER 2

In our current construction, the position map takes $O(n)$ storage space on the client, and the stash $O(\log N)$. Adding recursion to ORAM allows us to reduce the size of the overall client storage to $O(\log N)$, by reducing the space required for the position map to $O(1)$, while maintaining $O(\log N)$ for the stash (actually now multiple stashes). This is useful to us, because it allows us to implement statelessness. We can now write the client-side storage to the server after each call, and read it back again before the next one, without increasing the asymptotic bandwidth.

NOW WE'RE TALKING ABOUT THE IMPLEMENTATION, SO THIS SHOULD STAY

The essence of recursive ORAM, is that the position map of one ORAM is another ORAM. Thus, we can extend our original ORAM functor, parameterising it in the implementation of the position map. To do this we use a single interface, that is satisfied by both the in memory position map and the ORAM module. We can apply this new ORAM functor with the in memory position map module to get our original construction. However, we can now take this further, applying the functor again with the result of the first application, and this can be done to an arbitrary depth. Recursion has been made easy using the power of OCaml's module system.

There is of course hidden complexity here. How do we create a recursive ORAM? We definitely can't assume that the data ORAM can use the whole of the disk anymore, so we will have to do some calculations. How do we initialise recursive ORAM? How do we actually use ORAM as a position map? We will examine all of these now.

In particular talk about/show maths involved in automatically determining optimal size/number of levels of recursion

3.1.7 Statelessness

Do implementation of Statelessness!

Explain the implementation of Statelessness

3.1.8 Optimisation

Talk about optimising the block size through design space exploration

Talk about other optimisations that we're made as they come up

3.1.9 Integrity Verification

If had time to do integrity verification then include here otherwise discuss why I chose not to include it

3.2 File System

In order to perform search over documents, we need some way of actually storing those documents. This section describes the design and implementation of a basic file system that satisfies the requirements of the project.

3.2.1 General Design

The most common way of building a file system on top of a block device is through the use of inodes. An inode contains meta-information about a file along with pointers to the actual data blocks. For the purposes of this project, an inode will simply be one sector of the block device, containing the length of the file, followed by the list of pointers. In a system with more complex needs the inode would contain more information, such as modification/access timestamps, file permissions, etc., but we simply want to be able to read and write documents.

We need to be able to access the inode for a particular file quickly, so we should store its location in an index. We could perform lookup based on the actual filename, but the names have variable lengths, therefore, because we want to store the index, it is better to lookup based on the hash of the filename. Section 3.2.2 describes the implementation of the Inode Index.

We also need to allocate space on the block device for inode index blocks, for inode blocks and for data blocks. So we need a map that tells us which blocks on the device are free and allows us to update it as new blocks are needed. Section 3.2.3 describes the implementation of the Free Map.

We could now build a working file system, but we want it to be stateless. In order to do that, we need to store enough information to be able to reconstruct its in-memory representation. All we actually need to store is the root address of the Inode Index and the length of the Free Map. These two alone allow us to locate the data structures on disk when reconnecting to the block device. We store these two pieces of information at address 0 in what we call the Superblock.

3.2.2 The Inode Index

We need a data structure that associates keys, in the form of file hashes, with values, in the form of pointers to inodes. We want to support operations of insertion, lookup and deletion efficiently, but we also want to store the data structure on disk. This leads us naturally to an implementation using B-Trees.[1]

B-Trees are a generalisation of self-balancing binary search trees, where each node can have more than one child. If a node has n children, then it stores $n-1$ keys. It is guaranteed that

$$\forall k \in child_m, j \in child_{m+1}.k < key_m < j,$$

that is, a key is greater than all the keys to its left and less than all the keys to its right.

B-Trees are an efficient on-disk data structure, because we can use the whole of a disk sector for one node. This gives us an extremely high branching factor, reducing the depth of the tree and therefore the number of disk sectors that we need to access in any single operation. On creation of the file system, we calculate the branching factor of the tree in order to fill as much of each sector as possible with useful information.

3.2.3 The Free Map

In order to allocate space efficiently, we can simply use an array of bits the size of the block device. We again want to have an on-disk data structure, or at least one that can easily be flushed to disk regularly. It was therefore beneficial to write my own bit array based on `Cstruct`s, rather than using a library implementation. This gives us the ability to write the whole structure directly onto the disk using the block device methods, without any cumbersome translation.

The `Cstruct` library performs data access in bytes. This leads us to algorithm 4 for getting and setting individual bits. To get the n^{th} bit, we must get the $\frac{n}{8}^{th}$ byte and extract it from there. To do this, we calculate the index of the bit in the byte, shift a 1 to that position, and perform an and, masking that bit. Setting is a similar operation, but seeks to preserve the surrounding bits. To set a 1, we calculate the index of the bit in the byte, shift a 1 to that position, and perform an or, preserving all other bits. Setting a 0 is slightly trickier. We want to perform an and with a bit string that is 0 at the desired

[1] The algorithms for B-Tree operations were adapted from [1]

position and 1 everywhere else, but shifting fills empty bits with 0s. We can however use De Morgan's Law

$$a \;\&\&\; b = \neg(\neg a \;||\; \neg b)$$

to convert this to an operation involving a bit string that has a 1 at the desired position and 0s everywhere else.

Algorithm 4 Getting and setting individual bits in a byte array

```
function GetBit(index)
    byte ← byteArray[index]
    shift ← 7 − index mod 8
    return byte >> shift && 1
end function

function SetBit(index, boolean)
    byte ← byteArray[index]
    shift ← 7 − index mod 8
    if boolean then
        byte ← byte || 1 << shift
    else
        byte ← ¬(¬byte || 1 << shift)
    end if
    byteArray[index] ← byte
end function
```

3.3 Search Module

3.3.1 Inverted Index

Build Inverted Index

Talk about the implementation of the Inverted Index

Talk about the time constraints and the possible extensions that could otherwise be implemented

3.3.2 Keyword Search API

Design and implement a search API

Discuss the design/complexity trade-offs made

Discuss extensions to the search API that would be implemented in a more advanced IR system

3.4 Encryption

3.4.1 Library

Give an overview of the encryption library and talk about why it is better than rolling my own

Discuss the integration of the encryption library into the system

Listing 2 Mirage BLOCK module signature

```
module type BLOCK = sig

  type page_aligned_buffer = Cstruct.t

  type +'a io = 'a Lwt.t

  type t

  type error = [
    | `Unknown of string (** an undiagnosed error *)
    | `Unimplemented     (** operation not yet implemented in the code *)
    | `Is_read_only      (** you cannot write to a read/only instance *)
    | `Disconnected      (** the device has been previously disconnected
    ↪  *)
  ]

  type id

  val disconnect: t -> unit io

  type info = {
    read_write: bool;    (** True if we can write, false if read/only *)
    sector_size: int;    (** Octets per sector *)
    size_sectors: int64; (** Total sectors per device *)
  }

  val get_info: t -> info io

  val read: t -> int64 -> page_aligned_buffer list -> [ `Error of error |
   ↪  `Ok of unit ] io

  val write: t -> int64 -> page_aligned_buffer list -> [ `Error of error
   ↪  | `Ok of unit ] io

end
```

Chapter 4

Evaluation

4.1 Overall Results

4.2 Unit Tests

4.3 Performance Tests

4.3.1 Microbenchmarks

4.4 Security Analysis

Chapter 5

Conclusion

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[2] MS Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. *Network and Distributed System Security Symposium (NDSSâ12)*, 2012.

[3] Tarik Moataz, Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Resizable tree-based oblivious RAM. In *Financial Crypto*, 2015.

[4] Ling Ren, Christopher W Fletcher, Xiangyao Yu, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Unified Oblivious-RAM: Improving recursive ORAM with locality and pseudorandomness. *IACR Cryptology ePrint Archive*, 2014:205, 2014.

[5] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O(\log^3 N)$ worst-case cost. *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214, 2011.

[6] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

Appendix A

Project Proposal

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. In order to enable efficient search over the documents, Alice stores an encrypted index on the server along with the documents.

There are a number of schemes in the literature that use symmetric encryption techniques to build a searchable encryption scheme. They rely on the use of a trapdoor generating function, that allows Bob to search over the encrypted index and respond to Alice with the matching line from the encrypted index. Then Alice requests the relevant documents from Bob. Bob has a complete view of the communications channel, but does not have access to the trapdoor generating function. He simply sees a query in the form of trapdoor and then a number of requests for specific documents.

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 [2] on an encrypted email repository, up to 80% of plaintext 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. In our case, we move the searching and object retrieval functionality back to the client. 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, and therefore will never know which documents matched a given search query.

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 [6], 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.

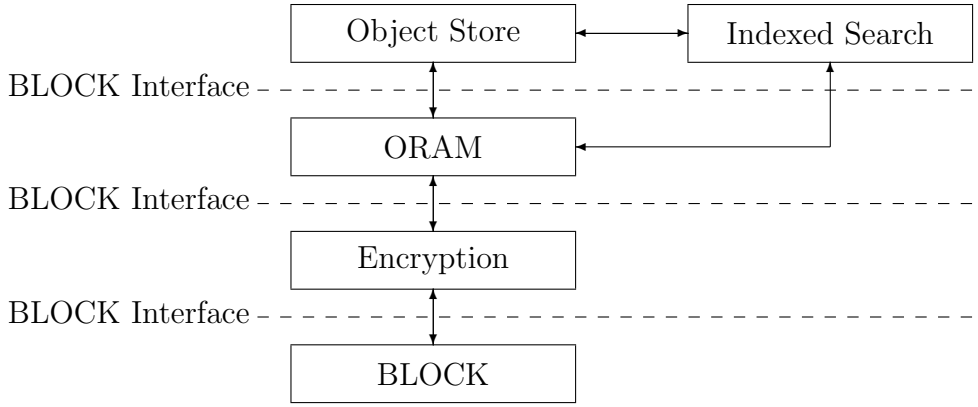

Figure A.1: The Application Stack: We can use any underlying BLOCK implementation and we can add/remove ORAM, Encryption or Search modules as we please

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 A.1.

Starting Point

MirageOS is a framework, that pulls together a number of libraries and syntax extensions, to provide a lightweight unikernel operating system, that is designed to run on the Xen hypervisor. A unikernel operating system is a single-address space machine image, customised to provide the minimum set of features to run an application. 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.

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 of static typing and a rich module system.

This will allow me to write my implementation of ORAM and Encryption as a pair of functors that take an implementation of Mirage's BLOCK interface and create new BLOCK implementations, augmented with new features. This means that we can add and remove ORAM and Encryption as we like and the Object Store remains agnostic. This is shown in Figure A.1. It also means that we could use any underlying implementation of the BLOCK interface and that it would plug seamlessly into existing programs. There are currently two implementations of the BLOCK interface, one for Unix and one for Xen, and I would like to support both. This abstraction also allows for the use of cloud storage, implemented as a mapping between the BLOCK interface and a cloud provider's RESTful API.

Other OCaml libraries that will be of most use to me include nocrypto, which provides

a wide variety of cryptographic tools, Jane Street's Core library, which standardises and optimises many of OCaml's core modules, and LWT, a lightweight cooperative threading library that is used throughout Mirage.

Substance and Structure of the Project

Substance

The main focus of this project is the implementation and evaluation of the Path ORAM protocol. Encrypted search is our target domain and as such will be an integral part of the project, but it is the performance and security properties that Path ORAM provides that we are really interested in.

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 stash, the accessed block is assigned a new random position and then as many blocks as possible are written back into the same path. The assignment to a random position 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 $ORAM_n$.

There is actually an issue with using Path ORAM in the context of MirageOS and cloud storage. If the Mirage instance crashes, then we lose the client-side state. With no position map, the ORAM becomes useless. To remedy this, we would have to read the entire contents out in one go and then reinsert it, resulting in a large overhead. There are two solutions to this problem: store the client-side state in persistent storage on the client, or upload the state to the server after every access. The second option is preferable, because it separates the ORAM implementation from the client machine. The client-side state is actually $O(\log N)$, so we should be able to store it on the server without increasing our complexity bounds.

As mentioned above, we can add integrity verification to Path ORAM by treating it as a Merkle tree. Each node will store a hash of the form $H = (b_1|| \cdots ||b_n||h_1||h_2)$, where b_n is the n^{th} block stored in the node, and h_1 and h_2 are the hashes of the left and right children. We always read and write the whole path at a time, so for the read or write of any single node we only have to read or write two hashes. For instance, on write, we

calculate the hash of the leaf node, which is then available for calculating the next level hash. So we only have to read the hash of the sibling of the leaf. This pattern is the same all the way up to the root of the tree.

In order to perform searches over our data, we will store along with it an encrypted inverted index. This is a data structure that, for any keyword, list the documents that contain it. The search module will build the index from the object store and then store the index using the object store. It will provide a search function that, given a keyword, will perform a simple scan over the inverted index and return the identifiers of documents that match it.

Evaluation

We need to test for functionality. Does the ORAM successfully write data and read it back out? Does the search function return documents correctly? This will consist of fairly trivial tests, writing objects to and from the block device and searching over them. A range of different types of documents will be used, including randomly (pre-)generated ones and entirely non-random ones, from sources such as Project Gutenberg.

We then want to evaluate performance. What is the overhead when we add the ORAM functor? How do recursion and statelessness further affect this? Does this correspond to the theoretical values from the literature? This will use tests similar to the above, but specifically focusing on time and space efficiency. Using the plain Object Store as a baseline, I will add in encryption and ORAM, separately and in combination to try and isolate the effects of each individual module.

Finally we want to test the security properties of the project. Is there any statistical correlation between access patterns? Do we provide adequate integrity verification? What, if anything, can be inferred about the search queries? Apart from integrity verification, which we can test by simply corrupting the ORAM and making sure that this is detected, the security comes down to the statistical independence of access patterns. If we can show, using statistical methods, that there is no correlation between two sequences of accesses with identical length, then we have security for not just storage, but for search as well, because this is protected by ORAM's security.

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 Object Store, testing this and further testing ORAM using it
4. Adding recursion and statelessness to ORAM
5. Implementing and testing the search module

6. Adding the encryption layer
7. Creation of a suite of tests and experiments to evaluate the performance and security properties of each individual component and of the system as a whole
8. Writing the dissertation

Success Criteria for the Main Result

1. To demonstrate, through well chosen examples, that I have implemented a functionally correct Path ORAM functor with search capabilities
2. To demonstrate, through well chosen examples, that the implementation has the expected security properties, i.e. keeps access patterns hidden

Possible Extensions

There are a number of ways that this project could be extended. By the nature of the modular design, we can perform optimisations at any layer of the system. There have been a large number of optimisations to Path ORAM proposed in the literature, so if I achieve the goals of my main project, including evaluation, ahead of schedule I will examine these and potentially implement some of them.

In particular for Path ORAM, 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.

Another 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 decrease for ORAM with small block size.

The last two optimisations were originally designed and tested in a secure processor setting [4], so their application to the cloud storage setting is novel.

Another area that could be addressed is the limitation of Path ORAM (and other tree based ORAMs) to fixed-sized trees. 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. Resizability has been implemented for the ORAM construction of *Shi et al.* [5] in [3], but exploring the possibility of making Path ORAM resizable was left as an open research topic.

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 object store.
3. **10/11/15 – 23/11/15** Finish object store and use it to build more complex tests of the ORAM.
4. **24/11/15 – 04/12/15** Add recursion and statelessness to the ORAM.
5. **05/12/15 – 18/12/15** Write up implementation section of the dissertation for all parts completed so far.
6. **18/12/15 – 31/12/15** Write the search module and design tests for it.
7. **01/01/16 – 08/01/16** Write implementation section for the search module.
8. **09/01/16 – 29/01/16** Evaluate the project in its current state, achieving an acceptably complete project. Write the progress report.
9. **30/01/16 – 08/02/16** Write up the evaluation of the project so far.
10. **09/02/16 – 21/02/16** Incorporate encryption model and perform further evaluation using this.
11. **22/02/16 – 06/03/16** Submit first draft to supervisors for feedback and modify based on feedback.
12. **07/03/16 – 11/03/16** Perform further evaluation and refinement as necessary
13. **12/03/16 – 25/03/16** Write final draft of dissertation and then leave it until submission time, in order to focus on revision.
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

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[2] MS Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. *Network and Distributed System Security Symposium (NDSSâ12)*, 2012.

[3] Tarik Moataz, Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Resizable tree-based oblivious RAM. In *Financial Crypto*, 2015.

[4] Ling Ren, Christopher W Fletcher, Xiangyao Yu, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Unified Oblivious-RAM: Improving recursive ORAM with locality and pseudorandomness. *IACR Cryptology ePrint Archive*, 2014:205, 2014.

[5] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O(\log^3 N)$ worst-case cost. *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214, 2011.

[6] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.