**Rupert Horlick**

# Encrypted Keyword Search Using Path ORAM on MirageOS

Computer Science Tripos – Part II

Homerton College

May 2, 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: | [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

# Chapter 1

# Introduction

## 1.1 Motivation

With cloud storage fast becoming ubiquitous, providers are faced with the challenge of guaranteeing the security of their clients' data. More than an exabyte of data was delivered by cloud storage providers in 2013 [1], and since so much of this data is held by only a handful of providers, trust is becoming a major concern.

Encryption might appear to be the solution to these trust issues; surely if the providers cannot read the plain-text of data then it must be secure. This seems to hold in general, but, in the application of query-based search, there is a problem. Islam et al. [6] demonstrated that using current methods of homomorphic encryption to search over encrypted documents can leak up to 80% of queries. Knowledge of the queries made to a data set, along with the number of documents returned by each query, could lead to dangerous inferences. As a motivating example, the discovery that a query to a medical database, such as $\langle name,\ disease \rangle$, returned results might allow an adversary to deduce information about a patient's medical status, constituting a breach of patient confidentiality.

Islam et al. [6] were able to infer search queries using the access pattern, the set of documents returned by each query. Thus, in order to protect against this kind of attack, we need to prevent the server from knowing which documents it returns in response to a query. Oblivious Random-Access Memory (ORAM) provides exactly that. Using ORAM, two accesses to the same piece of data, and, moreover, any two access patterns of the same length, are computationally indistinguishable to the server.

This project aims to demonstrate that, using Stefanov et al.'s Path ORAM protocol [15], it is possible to build a system that searches over 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

When dealing with security, the first challenge is to precisely define the threat model. The assumed capabilities of all parties must be clearly stated to ensure that security proofs are built on a solid foundation. The threat model for this project was refined a number of times, following the discovery of hidden assumptions, and is defined in Section 2.1.

Another challenge is taking a complex, abstract protocol, and making the design decisions required to turn it into a working system. The Path ORAM protocol abstracts away many implementation details, which had to be realised in this project.

Adding recursion to ORAM reduces the amount of client-side storage, making stateless ORAM more efficient. This presents us with the challenge of building recursive data structures, which are difficult to reason about and debug. This project takes advantage of OCaml's powerful module system, which separates the challenge of recursion from the underlying implementation.

The final challenge is choosing how to evaluate ORAM. There are many parameters and metrics that we could examine, however, due to the time consuming nature of running experiments on ORAM, this project limits its focus to the time overheads incurred by ORAM and the security of its construction.

## 1.3 Related Work

ORAM was first introduced by Goldreich and Ostrovsky [4] in 1996, who were motivated by the need for software protection on disks. The model was then expanded and refined for other settings, such as secure processors and cloud computing [13]. Stefanov et al. [15] made a significant contribution to the field, due to the simplicity and elegance of their protocol. Since then, many optimisations and additional features have been developed [17, 12, 8], and used to build a working, cloud-based storage system [14]. A useful thesis by Teeuwen [16] summarises the entire ORAM field, providing valuable insight into the evolution of ORAM.

# Chapter 2

# Preparation

## 2.1 Threat Model

This threat model involves three principals: the client, the server, and the attacker. The network is assumed to be under the control of the attacker. The attacker and the server are defined to be passive, and honest, but curious; they will both gather as much information as possible, without deviating from the protocol. Thus, the attacker will eavesdrop, but will neither prevent transmissions between the client and the server, nor tamper with, or produce their own, messages. The server will not tamper with the underlying storage. In this model, the attacker can be dealt with by the use of encryption, so it is the server that presents a threat, because it can see the access pattern of the underlying storage. The goal of this project is to ensure that this access pattern does not leak information from the search queries nor from the documents in the underlying storage.

## 2.2 Introduction to Path ORAM

The Path ORAM protocol is defined in terms of a client and a server, where the client stores data on the server. The data is divided into *blocks*, each of which is tagged with its offset in the data. We call this the block's *address*.

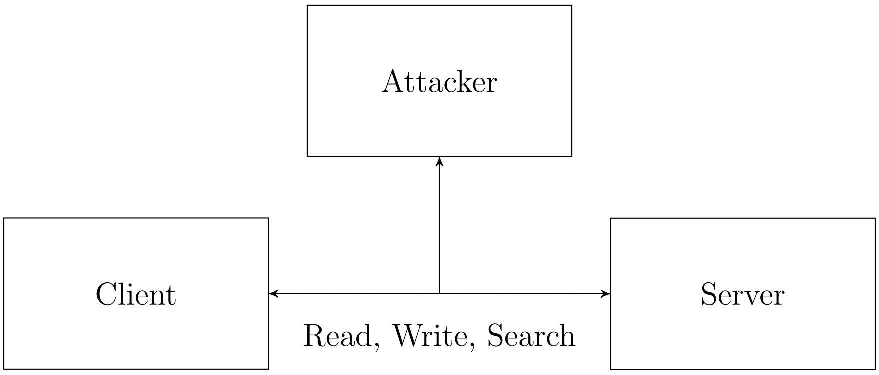


**Figure 2.1:** *The threat model: the attacker and server are passive, and honest, but curious.*

On the server, blocks are stored in a binary tree of height $L$. Each node in the tree is a *bucket* of size $Z$, that may hold up to $Z$ real blocks. Buckets must always be full, so dummy blocks are stored when there are fewer than $Z$ real blocks. Thus, the tree stores

$$N = Z \cdot (2^{L+1} - 1)$$

blocks in total.

The client stores two local data structures: the *stash* and the *position map*. The stash is a sort of working memory. As blocks are read from the server, they are written into the stash and may be returned to the server later on. Initially the stash is empty. The *position map* associates each address with a position between 0 and $2^L - 1$, which corresponds to a leaf, and therefore a path, in the tree. The position map is initialised by assigning a random position to each address.

The protocol maintains the invariant that, after the client performs a read or a write, a block with position $x$ is either in the stash, or in some bucket along the path to leaf $x$. This is achieved by using Algorithm 1 for both read and write operations. Its execution can be divided into four steps:

1. **Remap block**. The current position $x$ of the block with address $\mathsf{a}$ is read from the position map. A new position is chosen uniformly at random from $\{0, \ldots, 2^L - 1\}$ and is added to the position map.

2. **Read path**. The path to leaf $x$ is read into the stash. The block with address $\mathsf{a}$ will now be in the stash, if it has ever been written into ORAM.

3. **Write new data**. If the operation is a $\mathsf{write}$, the current block with address $\mathsf{a}$ is removed from the stash and is replaced by the new block containing $\mathsf{data}^*$.

4. **Write path**. The path to leaf $x$ is filled with blocks from the stash that meet the following condition. A block with address $\mathsf{a}'$ can be written into the bucket at level $l$ if the path to leaf $\mathsf{position}[\mathsf{a}']$ follows the path to leaf $x$ down to level $l$. If the number of blocks that satisfy this criterion is less than the bucket size, then the remainder of the bucket is filled with dummy blocks.

If a block in the stash is written back into the tree, then it must be in some bucket along the path to its assigned position. If not, then it is still in the stash, so the invariant holds after each execution of the algorithm.

The security of this algorithm is based on the random assignment of positions in step 1. If we consider the sequence of $M$ 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, as will two accesses to different addresses. Thus, by an application of Bayes' rule, we have

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

**Algorithm 1** Read/write data block with 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 $\mathsf{a}$ 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 $\mathsf{data}$
end function
```

i.e. the probability of the whole sequence is equal to the product of the individual probabilities, and therefore the access pattern is indistinguishable from a random sequence of bit strings.

To enable the client to disconnect from ORAM and reconnect later on, perhaps from a different machine, there must be no persistent client-side state. We call this *stateless* ORAM. This can be achieved by flushing the state, i.e. the stash and the position map, to disk after every operation. However, in the current model, the stash occupies $O(\log N)$ space, and the position map, $O(N)$, making this infeasible in practice.

*Recursive* ORAM reduces the space required by the position map to $O(1)$, taking the overall client-side state to $O(\log N)$. This means that statelessness no longer increases the asymptotic bandwidth overhead. Recursion stores the position map of the original ORAM, now referred to as ORAM$_0$, in another ORAM, ORAM$_1$. Thus, the client-side state becomes the position map of ORAM$_1$, along with the stashes of both ORAM$_0$ and ORAM$_1$. If each block in ORAM$_1$ can store $\chi$ positions then it will need $N' = N/\chi$ blocks. Therefore, the position map of ORAM$_1$ occupies $O(N/\chi)$ space. Repeating this recursion $\log N / \log \chi$ times leads to a position map of size $O(1)$. This sacrifices the space occupied by the recursive ORAMs and the time needed to perform recursive accesses, for efficient statelessness.

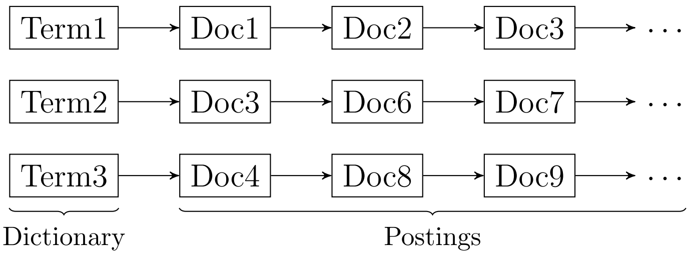


**Figure 2.2:** *The structure of an inverted index: the dictionary contains terms and the postings contains lists of documents that each term appears in.*

## 2.3 Introduction to Inverted Indexes

The *inverted index* is the most important data structure in Information Retrieval. The amount of time required to perform a query is reduced by performing a large amount of work in advance. A simple linear scan of $N$ documents takes $O(N)$ time. Using an inverted index also takes $O(N)$ time in the worst case, but the constant is greatly reduced and empirically the index delivers excellent performance.

The index 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. Each term has a postings list, a list of all the documents that contain that term. Collectively these postings lists are referred to as the postings.

An inverted index can be constructed in three steps:

1. Split each document into *tokens*. These are units of the document separated by spaces.

2. Perform linguistic preprocessing on the tokens. An example is *stemming*, which removes suffixes of words, converting them into a normalised form.

3. Assuming each document has an ID, add the ID to the postings list of each token the document contains.

Look-up of a single keyword in a hash-based inverted index is performed by hashing the keyword and returning the relevant postings list, if it exists. Simple Boolean operations can be added. For instance, disjunction takes the union of two postings lists, and conjunction takes the intersection.

This project will limit its focus to conjunctive queries, as it aims to demonstrate the correctness and efficiency of search using the ORAM implementation, rather than creating an advanced IR system. Queries are space-separated lists of keywords, and the result of a query is the conjunction of the postings lists of all keywords it contains.

## 2.4 Introduction to MirageOS

Running ORAM in the cloud allows a user to access their data from any location. The ORAM client is a trusted cloud instance, and the server is a cloud storage provider, as illustrated in Figure 2.3.

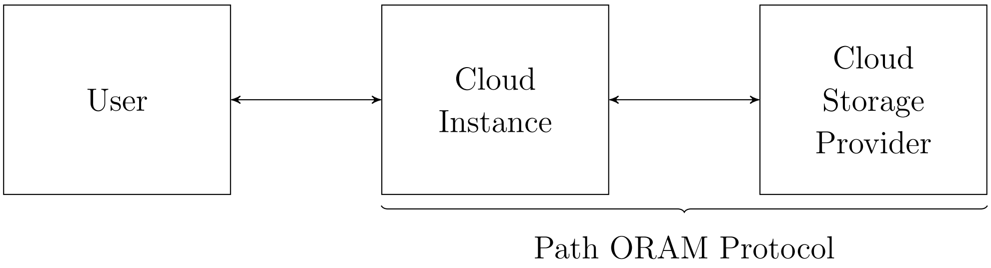


**Figure 2.3:** *ORAM can be built as a MirageOS application, which can run on a trusted cloud instance.*

MirageOS is a unikernel operating system. In other words, a MirageOS application is compiled to an executable, along with only the necessary parts of the OS. It can be compiled for a number of targets, including Unix or Xen. An executable, running directly on the Xen hypervisor in the cloud, is more lightweight than the traditional cloud stack, which runs an application on a full operating system, such as Ubuntu. By building ORAM as a MirageOS application, this project allows an instance running ORAM to be spun up whenever the user needs, and shut down when not in use, minimising its cost.

## 2.5 System Architecture

This project uses the framework illustrated in Figure 2.3. The focus is on implementing the code for the cloud instance, which consists of the implementation of Path ORAM, a file system running on top of it, and a search module that presents an API to the user. Writing interfaces for specific cloud storage providers is left as future work, so for the purposes of this project a local block device is used for storage.

MirageOS provides an interface for a block device, `BLOCK`. The ORAM module is designed to satisy this interface, so that it can be inserted into exisiting Mirage applications. It is also designed to run on any underlying storage that satisfies `BLOCK` and, thus, interfaces for cloud storage providers could be written to be compatible with ORAM. The encryption module is designed in the same way, so it can be inserted between ORAM and the underlying storage. Figure 2.4 shows the overall structure of the application.

## 2.6 Requirements Analysis

**High Priority** Basic ORAM

**Medium Priority** File system, Search module

**Low Priority** Encryption, Statelessness (Extension), Recursion (Extension)

Building ORAM is the core focus of the project and as such is given high priority. The addition of the file system and search module creates a complete system that can be evaluated, so these two modules are of medium priority. Encryption is necessary in a real world system, but not to perform evaluation, so it is given low priority. Recursion and stateless are extensions, and are therefore deemed to have low priority.

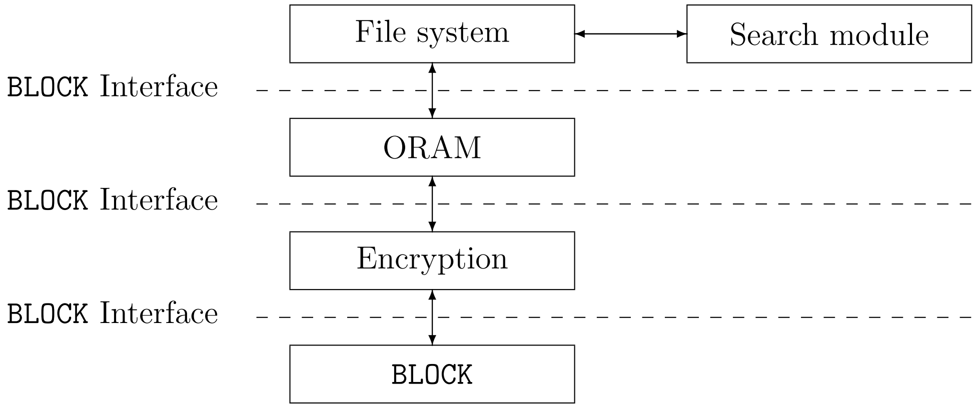


**Figure 2.4:** *The application stack: ORAM satisfies MirageOS's* `BLOCK` *interface and any underlying* `BLOCK` *implementation can be used.*

# 2.7 Choice of Tools

## 2.7.1 OCaml

OCaml is the logical choice of programming language when building for MirageOS, because Mirage applications and libraries all use OCaml. However, OCaml's module and type systems were also considerations in the choice of Mirage for this project.

OCaml has a powerful module system. It allows one to build *functors*, which are modules parameterised over module interfaces. Any module that satisfies the interface can be given to the functor to create a new concrete module. ORAM can therefore be implemented as a functor that is parameterised over the `BLOCK` interface.

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

## 2.7.2 Libraries

The libraries used for building ORAM are listed in Table 2.1.

| *Library* | *Version* | *Purpose* | *License* |
|---|---|---|---|
| Mirage | 2.6.1 | System Component Interface Definitions, Application Configuration Framework | ISC |
| Jane Street's Core | 113.00.00 | Data Structures, Algorithms | Apache-2.0 |
| LWT | 2.5.1 | Threading | LGPL-2.1 |
| Cstruct | 1.8.0 | Data Structure | ISC |
| Alcotest | 0.4.6 | Unit Testing | ISC |
| Mirage Block CCM | 1.0.0 | Encryption | ISC |
| Stemmer | 0.2 | Linguistic Processing | GNUv2 |

Table 2.1: Libraries used by Mirage ORAM

| *Tool* | *Version* | *Purpose* | *License* |
|---|---|---|---|
| Mac OSX | 10.11.2 | Operating System | Proprietary |
| Emacs | 24.5 | Text Editor | GPL |
| git | 2.8.0 | Version Control | GPLv2 |
| 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

### 2.7.3 Development Environment

Choosing the right tools is essential to the productivity of large projects. For OCaml, one of the most important tools is OASIS, which automatically generates Makefiles for a project, based on a specification file. Direct use of the OCaml compiler quickly becomes infeasible when linking together a large number of modules. OASIS deals with this automatically. Emacs proved incredibly useful, because code indentation, syntax highlighting, autocompletion, and type inspection are all provided by third-party plugins. Code can be interpreted, compiled, and run from a shell within Emacs, which boosts productivity.

## 2.8 Software Engineering Techniques

I employed two key techniques to ensure my code was well-designed and well-built, without compromising productivity.

Firstly, I wrote the interface file for each module before writing the implementation. This forced me to make important design decisions up front, clarifying the structure of the module and its relation to the system as a whole.

Secondly, I practised Test Driven Development [5]. I unit tested each new piece of code as it was written, allowing me to fail fast and fix problems at their source. This approach meant that small modules could be integrated into a larger system that worked as expected more often than not.

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

## 2.9 Summary

This chapter has covered the work undertaken prior to development. This included a definition of the threat model, a brief introduction to the major algorithms, data structures and libraries, an overview of the preliminary architectural design, and a discussion of the techniques and tools selected for the development process.

The next chapter demonstrates how this preparatory material was applied to successfully implement the Path ORAM protocol on MirageOS, and how this protocol was used to build a secure encrypted keyword search application.

# Chapter 3

# Implementation

This chapter explains the process of building a functioning system, using the designs and algorithms of the previous chapter. An overview of the system is given in Figure 3.1. Each module will be examined in turn, working upwards through the diagram. Thus, the chapter begins with a discussion of encryption in Section 3.1, followed by a longer discussion of ORAM in Section 3.2, which constitutes the main focus of the project. This includes subsections about the extensions: recursion and statelessness. Finally, the file system and search modules are explored in Sections 3.3 and 3.4 respectively.

The main challenges and achievements of the implementation can be summarised with reference to Figure 3.1.

At the inter-modular level, integration of all parts of the system required careful API design and intricate manipulation of the OCaml module system.

For encryption, an appropriate library had to be chosen, a process that involved filing a pull request to fix a critical bug.

The ORAM module presented the challenge of translating the terse pseudocode of the Path ORAM protocol into a functioning program. This included designing a position map capable of operating on machines of any word size, and building functions to marshall data to and from the format required by ORAM. Adding recursion to this implementation warranted a deeper understanding of the module system, including first class and recursive modules. To achieve statelessness I had to serialise recursive data structures, which meant writing custom functions to conform with a binary protocol.

The implementation of a minimal, but complete, inode-based file system required investigation into the choices made by many file system designers before me. I weighed these against the demands of my system and included only the necessary elements. Furthermore, my file system entailed an implementation of B-Trees, which did not previously exist in OCaml. I therefore implemented a B-Tree library myself that can be applied to other Mirage applications, representing a contribution to the community.

Finally, the search module made use of concepts from the field of Information Retrieval, including algorithms and data structures. The decision to include stemmming represented a significant trade-off between the space used by the indexing process and its precision.

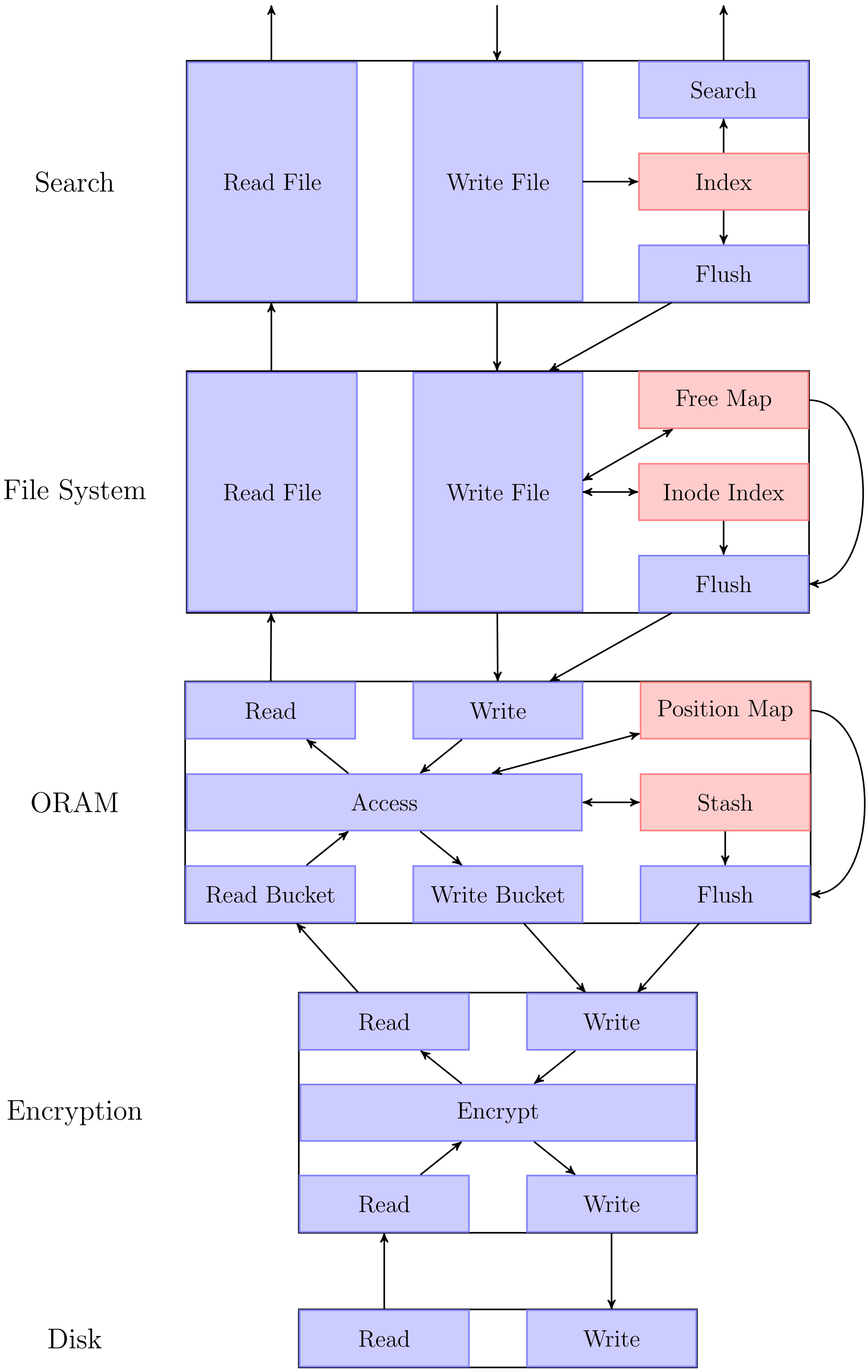


**Figure 3.1:** *An overview of the system, showing the flow of data. Black boxes are modules, blue are functions, and red are data structures.*

**Listing 1** 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
```

## 3.1 Encryption

The security of our entire system is dependent on the security of the underlying encryption layer. Without encryption the ORAM construction is completely superfluous, thus we want to use tested and trusted cryptographic libraries, rather than rolling our own. Luckily for us, there is an existing implementation of an encrypted block device satisfying Mirage's BLOCK interface. It provides a functor, a module parameterised in another module. This functor takes an implementation of a block device, satisfying the BLOCK interface, and returns a new module, also satisfying the same interface. This is very similar to the way that ORAM is implemented, and allows us to chain together the application of this functor with the ones used in ORAM to get an encrypted block device. To create the encrypted block device and later connect to it, we need to provide a key. Key management is a very important problem, but it is left out of the scope of this project. For our purposes we use a constant, known key throughout the evaluation stage of the project.

## 3.2 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.

### 3.2.1 Inherent Constraints

By virtue of writing an implementation that satisfies 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, introduced in Section 2.7.2. The underlying block device requires a buffer of type `Cstruct.t` to read from/write to, and in order to satisfy the `BLOCK` module signature, ORAM needs to return data as a `Cstruct.t`. Thus, to avoid unnecessary marshalling of data, 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 possible.

### 3.2.2 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.

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 Stefanov et al. [15], we require exactly $Z \cdot (L + 1)$ blocks of transient storage, for a tree of height $L$ and bucket size $Z$, 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, $\lambda$, and the bucket size $Z$. A stash with security parameter $\lambda$ 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, but we want our security parameter to be high enough that we have long term security. We must make a trade-off between security and speed, which we can do by keeping the security parameter high and reducing the size of a block. As we have just seen there is a constant maximum number of blocks in the stash, so reducing the block size will have a direct affect on the time taken for each access.

| Security Parameter ($\lambda$) | 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 from Stefanov et al. [15]

### 3.2.3 Position Map

The position map associates a leaf position with each block of the data. As mentioned in Section 3.2.1, we are constrained to using `int64` block addresses, thus we require 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 $2^{64} - 1$, using a type that can only go up to $2^{30} - 1$ on a 32-bit machine.

To accommodate this, we can 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 consists of the 4 most significant bits, and will therefore have a value of 0 unless we store more than $2^{60}$ blocks. The 30-bit values are guaranteed to be converted into non-negative `int`s, which we can then use to address two dimensions of the array.

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.

### 3.2.4 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 1, 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 2. This type contains the ORAM parameters such as `bucketSize` and `blockSize`, 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:

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

```
Require: $\mathsf{size} > 0$

function PosMapDims(size)
    $(x, y, z) \leftarrow$ SplitIndices(size)
    $x \leftarrow x + 1$
    $y \leftarrow y + 1$
    if $x > 1$ then
        $y \leftarrow$ 0x3FFFFFFF
        $z \leftarrow$ 0x3FFFFFFF
    else if $y > 1$ then
        $z \leftarrow$ 0x3FFFFFFF
    end if
    return $(x, y, z)$
end function
```

`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. The `BLOCK` interface defines the size of the block device in sectors, using the variable `size_sectors`, and defines the size of a sector using `sector_size`. We will continue to refer to data blocks in ORAM as blocks, but to satisy `BLOCK`, we will need to expose a value for `sector_size`. 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. Now, the `sector_size` that ORAM uses is the size of the part of the block that stores data. Thus, we must subtract the size of the address, 8 bytes, giving

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

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

$$L = \left\lfloor \log_2 \left( \frac{N}{Z} + 1 \right) \right\rfloor - 1.$$

This comes from rearranging the equation for the size of the binary tree in buckets, $2^{L+1} - 1$, introducing the floor operator so that the resulting binary tree is less than or equal to the desired size, so that it will definitely fit on the block device. If the size

**Listing 2** The type of an ORAM device `ORAM.Make(B).t`

```
type info = {
    read_write: bool;
    sector_size: int;
    size_sectors: int64;
  } [@@ deriving bin_io]

type structuralInfo = {
    height : int;
    numLeaves : int64;
    sectorsPerBlock : int;
  } [@@ deriving bin_io]

type core = {
    info : info;
    structuralInfo : structuralInfo;
    bucketSize : int64;
    offset : int64;
    desiredBlockSize : int;
    stash : Stash.t;
  } [@@ deriving bin_io]

type t = {
    info : info;
    structuralInfo : structuralInfo;
    bucketSize : int64;
    offset : int64;
    desiredBlockSize : int;
    stash : Stash.t;
    positionMap : PositionMap.t;
    blockDevice : BlockDevice.t;
  }
```

is unspecified, we assume that we should fill as much of the device as possible, so we calculate

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

and then perform the same calculation as above with this new value. Finally we calculate, `numLeaves` and a new value for `size_sectors` from $L$ using the equations we have already seen.

This is all of the structural information we require, so finally we must create instances of the client-side data structures and we must initialise the ORAM space. To do the former we call the creation functions of the associated data structures. For the latter, we loop 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`.

### 3.2.5 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.ts`, so they must split this into chunks before calling `access` on each one. On the other side

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

```
function BucketAddress($x$,$l$)
    $address \leftarrow 0$
    for $i = 0;\ i < l;\ i++$ do
        if $x >> (i + \texttt{height} - l)\ \&\&\ 1 = 1$ then
            $address \leftarrow (2 \times address) + (\texttt{bucketSize} \times \texttt{sectorsPerBlock} \times 2)$
        else
            $address \leftarrow (2 \times address) + (\texttt{bucketSize} \times \texttt{sectorsPerBlock})$
        end if
    end for
    return $address$
end function
```

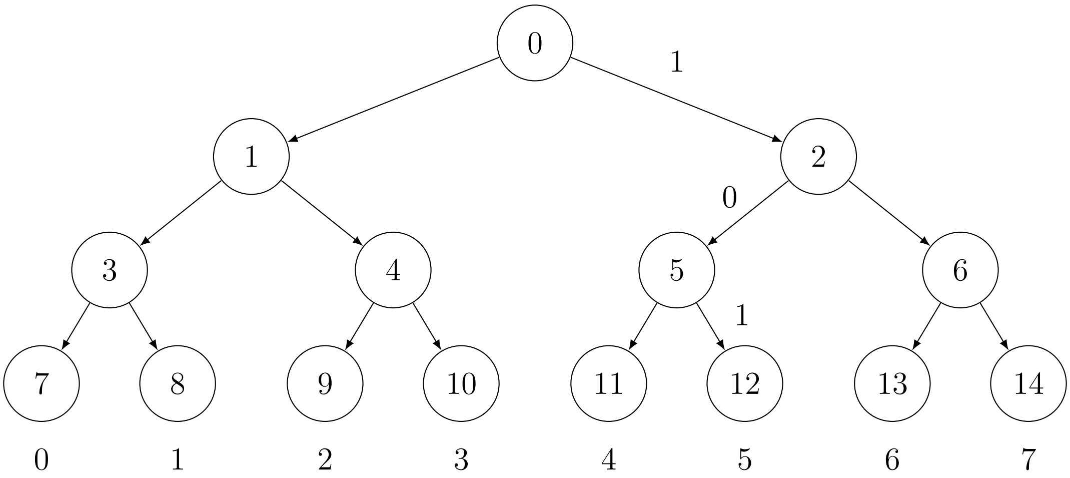


**Figure 3.2:** *Visualisation of Algorithm 3*

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.

This is most easily explained using Figure 3.2. 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.

With this plumbing in place, we can input/output data from the outside and the underlying block device. Now we can implement Algorithm 1. Remapping the blocks requires a call to a pseudo-random function, reading in the path adds the contents of each bucket to the stash, and writing new data looks up a block in the stash and gives it a new data value. The final part 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 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 perform a right bit shift on both $x$ and $\mathsf{position}[\mathsf{a}']$ of $\texttt{height} - l$ bits, and check for equality.

We now have a functioning ORAM functor, that can be used to augment a block device in any Mirage program. In Sections 3.2.6 and 3.2.7 we look at optimisations and extensions to this ORAM construction, before moving onto to the further aims of the project.

### 3.2.6 Recursion

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. This is an exhibition of the power of OCaml's module system.

We can use the method above to build the recursive ORAM module by hand, applying the functor $n + 1$ times for $n$ levels of recursion. However, it is better to have a more automatic solution, where the recursive module is built automatically based on the size of the data ORAM, $\mathrm{ORAM}_0$. Our addresses are 64-bit integers, so they take 8 bytes of storage each. If $\mathrm{ORAM}_0$ has size $N$ in blocks, and the size of each block is $B$ bytes, then one block in $\mathrm{ORAM}_1$ being used as the position map for $\mathrm{ORAM}_0$ can store $\chi = B/8$ addresses. The number of blocks required for $\mathrm{ORAM}_1$ will therefore be $N/\chi$. After $\log N/\log\chi$ recursions, we will have a constant sized position map.

So, to create a recursive ORAM, we take in the size, $N$, and the block size, $B$, and automatically apply the ORAM functor recursively $\log N/\log\chi$ times. Calling the `create` function of the resulting module will create ORAM instances with the correct number of levels of recursion.

### 3.2.7 Statelessness

In order to achieve statelessness for ORAM, we need to store type information, the stash, and the position map. We will first talk about how to layout all of this information on disk, before moving on to talk about how we actually get it there.

Once we have initialised ORAM and begun to use it, we do not want to move it around on disk, because we would have to copy the whole data structure. We do however need to store the information necessary to discover ORAMs existence in a well-known location, in order to allow us to reconnect to it. This leads us to storing a what is called a superblock in the first block of the underlying block device. This block will contain a pointer to the location of the rest of the state, along with its length. This way, we can store ORAM starting at the second location on disk, and then append all of the state at the end of the ORAM section, meaning we will never have to move ORAM once we have initialised it.

Now we actually need to store the state, which means we need some method of serialising it, ready to be written to disk. We could implement this manually, but for the majority of the information we can take advantage of an existing serialisation library, Jane Street's Bin_prot. This is a binary protocol, that allows you to annotate a type with `[@@ deriving bin_io]` and will then generate functions to read and write instances of the type into buffers. We are able to use this for all of the type information for ORAM, as well as for the stash. This leads us to splitting the ORAM type into a core, that can use Bin_prot, and an extended type, that includes the position map and the underlying block device. This structure is shown in Listing 2.

The position map is more difficult to serialise, because under recursive ORAM, it might actually be another ORAM. We don't want to write the entire of the position map ORAM onto the disk a second time, so we need a custom serialisation function for the position map that will store only meta-data for ORAM position maps, and will store the entire position map in the base case. We want to store all state right at the end of the block device, after all recursive instances, so this function actually collects the data from all the levels of recursion together into one buffer.

Now that we can write all of the major parts into the buffer, we collect everything together and flush it to the end of the block device. After this is done, we are able to disconnect from ORAM safely. Reconnecting to ORAM is now a case of checking for the presence of the superblock, reading in the location and length of the state, reading the actual state, and then calling the connect function on the position map. The connect function allows each recursive ORAM instance to have its own reference to the underlying block device and returns once it reaches the in memory position map.

## 3.3 File System

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

### 3.3.1 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. In a real system we would need to deal with collisions in the hash function, but for a small number of files we can neglect this possibility. Section 3.3.2 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 free map. This is a map that tells us which blocks on the device are free and allows us to update it as new blocks are needed. Section 3.3.3 describes the implementation of the free map.

We also want our file system to be stateless, and in order to do that, we need to store the data structures, along with enough information to find them on disk. We only need to store the root address of the Inode Index and the length of the Free Map to locate the data structures on disk when reconnecting to the block device. We store these two pieces of information at address 0 in another superblock.

### 3.3.2 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

$$\forall m \leq n, k \in \mathit{child}_m, j \in \mathit{child}_{m+1}. k < \mathit{key}_m < j,$$

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 block for one node. This gives us an extremely high branching factor, reducing the depth of the tree and therefore the number of blocks 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 block as possible with useful information.

[1] The algorithms for B-Tree operations were adapted from Cormen et al. [2]

**Algorithm 4** Getting and setting individual bits in a byte array

```
function GetBit(index)
    $byte \leftarrow \mathsf{byteArray}[\mathsf{index}]$
    $shift \leftarrow 7 - \mathsf{index} \bmod 8$
    return $byte >> shift \;\&\&\; 1$
end function

function SetBit(index, boolean)
    $byte \leftarrow \mathsf{byteArray}[\mathsf{index}]$
    $shift \leftarrow 7 - \mathsf{index} \bmod 8$
    if boolean then
        $byte \leftarrow byte \;||\; 1 << shift$
    else
        $byte \leftarrow \neg(\neg byte \;||\; 1 << shift)$
    end if
    $\mathsf{byteArray}[\mathsf{index}] \leftarrow byte$
end function
```

### 3.3.3 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 `Cstructs`, 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 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.

## 3.4 Search Module

So we have our documents and we can access them without revealing which ones we are accessing. Now the final piece of the puzzle is actually performing search. We discuss building an inverted index, the data structure that will allow us to search efficiently, in Section 3.4.1. We then discuss the front-end of the whole application, the search API, in Section 3.4.2.

### 3.4.1 Inverted Index

The basics of inverted indexes are discussed in Section 2.3. As stated there, the index consists of two main structures, the dictionary and the postings. For the dictionary, we will use the most common implementation, a hash table. This provides us with $O(1)$ lookup and insertion, which are the main operations we will be performing. The postings are more flexible. In our implementation we want to store the file names, because they are not actually stored in the file system itself. We also want to perform intersection of postings lists, allowing us to perform conjunctive queries. Thus, we will use a hash set, a data structure built on top of a hash table, that stores a set of keys, and has the added benefit of keeping them unique for us.

Having decided on our data structure, we need to actually index files. As usual in this project, the files will be `Cstructs`. So we have a number of steps to perform in order to process a file. We first convert the file to a string and immediately strip it of characters we do not need, including all punctuation. At this point we have a sequence of alphanumeric character strings, separated by spaces and newlines, so we can perform a split on these characters to get a list of words.

We could perform indexing now, adding the name of the current file to the postings list of every word in our list, but there are a couple of things that we want to do first. We do not want to store separate words for 'run', 'ran', 'runs', and so on, so we will perform some linguistic preprocessing. We will perform stemming, which uses a set of rules to prune suffixes, mapping words onto a stem. We will do this using Porter's stemming algorithm [10]. I used a small open-source library implementation of this algorithm, ocaml-stemmer. This technique not only reduces the size of the index, but also arguably improves search, because now queries for 'run' can automatically returns documents containing morphological derivations. Finally, we remove duplicates from our list of words, reducing insertion overhead, and put the entries into the index.

For our purposes we will only implement simple conjunctive queries, meaning we look for documents containing all of the words in a space separated query string. Now to search we perform the same preprocessing on the query string, performing lookup on each of the queries in turn and taking the big intersection of the resulting list of hash sets. In order to make this intersection operation efficient, we sort the list by order of hash set size. Then we can filter the smaller hash set by checking for membership in the larger hash set. This means we perform one constant time lookup for each member of the small set, rather than the large, giving a large performance boost, as this smaller set is monotonically decreasing.

### 3.4.2 Keyword Search API

The final step in an end-to-end system is the API. We need to wrap file system access, indexing and search all into one module that provides a single point of entry for an encrypted search system.

We have three main operations that are the most important to support. Writing files, reading files, and searching over files. We will not be concerned with deleting files in this project, because they are not important for the evaluation of ORAM.

Writing files is the most important step, because this is where the search module does the indexing. On write, we first write through to the file system, then index the file and finally flush the index to disk to make sure it persists. Reading files is simply a pass through to the file system and search makes calls to inverted index.

## 3.5 Summary

In this chapter I have discussed the implementation of all of the components of my system, including the important design decisions and trade-offs that were made. We have seen how to implement recursive Path ORAM in a way that allows it to plug into existing MirageOS programs, and we have seen how to build a file system and search operations on top of it.

We now move on to the important task of evaluation, where we make sure that I have implemented ORAM in a way that ensures functionality, performance, and security, and have therefore achieved the aims of the project.

# Chapter 4

# Evaluation

This chapter explains the methodology used to ensure the correctness of the ORAM implementation, as well as analysing its performance and security properties. Section 4.1 discusses functional testing through the use of unit testing and randomised testing. Section 4.2 then goes on to discuss performance, and then finally we analyse the security of ORAM in Section 4.3.

Overall, ORAM performed as expected in terms of functionality, performance, and security. It operated correctly, writing files and reading the same data back out. It continued to do so with the additions of statelessness, recursion, and encryption, so all parts of the system were functional separately and as a whole. In terms of performance, my implementation agreed with the theoretical bounds given in Stefanov et al. [15], which state $O(\log N)$ time overhead. Finally, statistical analysis showed that ORAM did indeed have a statistically random access pattern, ensuring the security of the implementation.

If I had had more time, I would have performed more evaluation, examining different parameters of ORAM such as bucket size, extending experiments to larger ORAM sizes, and analysing different metrics such as bandwidth usage. Unfortunately experiments on ORAM take in the order of a few days to initialise and then a further day to run, so despite having a large amount of time to perform experiments, the nature of them restricted the amount that I could achieve.

## 4.1 Unit Tests

Unit testing was used throughout the development process, which allowed me to be sure that individual components were functioning correctly, before I combined them into larger more complicated systems. This section describes the most important test cases that were examined for each module and discusses the use of randomised testing to cover a larger range of input values. Code coverage testing was also used to make sure that the tests were visiting all parts of the system.

### 4.1.1 Stash

There are three main cases to test for the stash:

- Values not expected to be in the stash are not found
- Values that have been added to the stash are found
- Adding dummy blocks to the stash has no effect

During the development process, I hand-coded a small number of example cases, but in order to test more extensively, I coded the above three cases as properties for randomised testing. This generates a number of test cases and verifies that the properties hold in every case and allows us to cover far more cases than hand-coded tests alone.

### 4.1.2 Position Map

We need to test the two main aspect of the position map. Firstly, the translation of 64 bit addresses into 3 regular integers, and then its actually operation as a data structure.

The code performing the translation is essentially a mathematical definition in itself, so any property definition that we might use for randomised testing would probably just be the original code. Thus, in this instance we simply check a few hand chosen random cases, along with the important edge cases. The edge cases we want to check are the maximum, and minimum values, along with values either side of a change in the higher indices, that is values with output $(x, y, max\_int)$ and $(x, y+1, 0)$, and similarly for the higher index.

In terms of operation, we want to check that on adding a value to the position map, we read back the same value, and that trying to add a value at an address outside of the allowed range results in an error. Both of these cases can be coded up as properties for randomised testing.

### 4.1.3 ORAM

The ORAM implementation is particularly amenable to randomised testing, because we tend to define inverse functions for most of its functionality. This makes it very easy to write properties for randomised testing of the form $f(f^{-1}(x)) = x$. This allows us to check that each stage of ORAM is operating correctly, from writing individual blocks, through writing entire paths, to writing entire files.

For other functions that were not so easy to code properties for, regular unit testing was used. These include reconnecting to ORAM to test statelessness, and performing simple calculations, such as computing the height for ORAM.

### 4.1.4 Free Map

The Free Map allocates blocks in the block device to be used by different parts of the file system. We need to make sure that on creation, it allocates the correct number of initial blocks, and then subsequently always allocates the correct number of blocks, if they are available, and only ever allocates blocks that are actually free.

During development, I used only a handful of test cases, covering some of the most important edge cases. This included creating the map, and checking that nothing was allocated to begin with except the first $n$, where $n$ is passed into the creation function. Then, I checked that allocating and deallocating a few different sequences of blocks led to the expected results. Finally, I checked that if there were not enough free blocks, that an error was returned.

After main development had finished, I added some randomised testing to this module to cover a wider range of inputs.

### 4.1.5 B-Trees

It is not possible to test the B-Tree library directly, because it only gives us a functor to create a B-Tree. Thus the B-Tree was tested using the Inode Index, the canonical use case of the library in this system. In order to test the B-Tree properly, it was necessary to use randomised testing to create a reasonable access pattern that was guaranteed to cause splitting of the root node.

### 4.1.6 Inodes

The inodes needed to be tested to make sure they could add and delete pointers, while maintaining a correct count. This is again amenable to randomised testing, although unit tests were also used throughout the development to test specific cases that would be expected to cause exceptions.

### 4.1.7 File System

In order to test the file system I created a large number of random files and used randomised testing to ensure that under any access pattern the correct files were always read back out once they had been written in. This also included testing to ensure that reasonable exceptions were produced when the file system became full, or a file did not exist in the system.

### 4.1.8 Search Module

The search module was tested for correctness, i.e. if a file containing a word had been put into the system, then it would appear in the search results. This was difficult to randomise, so I manually constructed a number of test scenarios.

## 4.2 Performance Testing

### 4.2.1 Parameter Optimisation

Before performing the main experiments, I decided to optimise the parameters of ORAM, in order to allow the experiments to run faster. The main parameter in question is the block size, which has been shown in Ousterhout et al. [9] to dramatically affect the speed of IO operations. I discovered that this was indeed the case with ORAM, as can be seen in Figure 4.1. It appeared that increasing the block size has an unbounded increase on

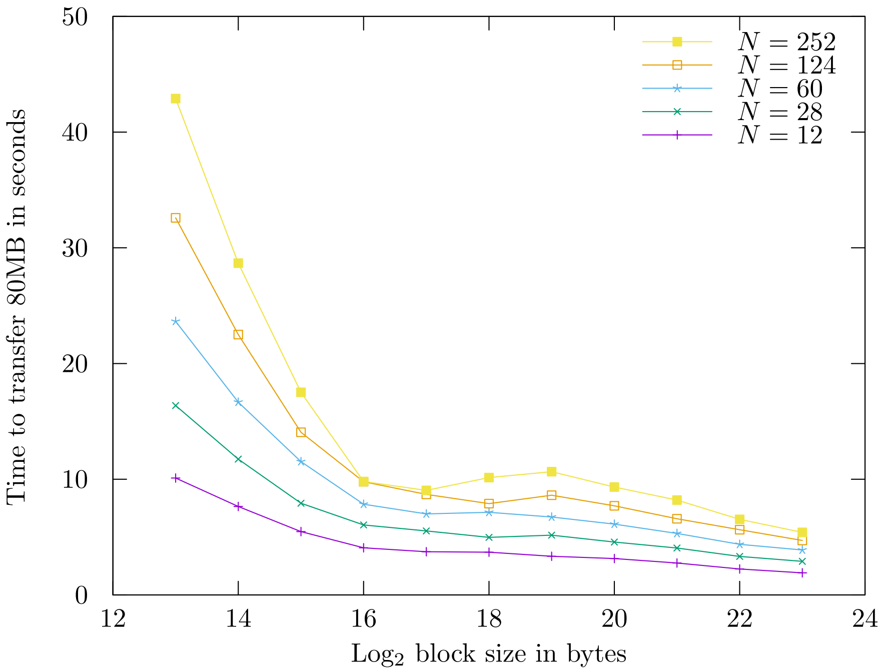


**Figure 4.1:** *Plot of the time taken to transfer 80MB of data at varying block sizes and sizes of ORAM. Each line represents one ORAM size, so we can see that as block size increases, the time decreases.*

performance, but I settled with a block size of 1MB to trade-off between speed and the possibility to specify the size the block device.

It is important to note that using this in the cloud, we would see very different results. Increasing the block size will increase the size of the stash proportionally, because the maximum stash size is a constant number of blocks. Thus, a larger block size increases the amount of data that we must write to disk in order to achieve statelessness. We have shown that increasing block size increases speed when using a local disk, but network latency will dominate when running in the cloud, leading to a slow down. I would need to perform further experiments in the cloud to discover an optimal trade off point for this scenario.

### 4.2.2 Comparison with Literature

In order to effectively test the overhead due to ORAM, two things needed to be done very carefully. The first was isolating and removing major sources of uncertainty. When I first ran the experiments, I attempted to run them on my local machine. In this environment, other processes interfered with the ORAM process, making the results unreliable. I secured a remote testing machine in order to run the experiments in complete isolation. Here, at first, they were running on an NFS mounted drive, meaning that network latency and protocol overheads were affecting the results. After moving the experiments to local disk, it was clear that ORAM overheads were finally dominating the results. Now, secondly, I needed to initialise the ORAMs properly. Using a fresh ORAM, the stash is empty and most of the blocks are dummy blocks that get disregarded. It is necessary to

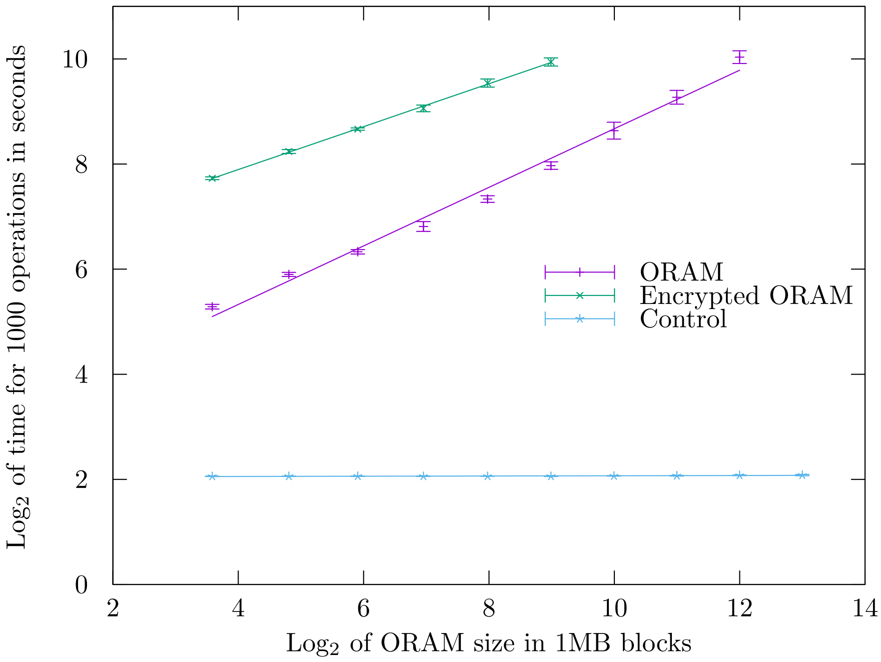


**Figure 4.2:** *The relationship between size of an ORAM in blocks and the time taken for 1000 operations, plotted for ORAM, encrypted ORAM, and a control block device with no ORAM. We take logs of both axes, because block size was increased in powers of two and we expect a log relationship.*

run an initialisation sequence in order to remove the effects of these on the results. We use the worst case sequence, writing each block in turn, then reading each block in turn, to ensure that all blocks get used multiple times, leaving the ORAM in a state that it would likely be in after extended use. Only once this steady state has been reached can we reliably test the performance.

For the actual experiment, rather than using the worst case sequence, we use a random sequence of block accesses. We perform 10 runs of 1000 iterations each, oscillating between reads and writes in order to balance the different overheads. We are trying to see how the performance changes as the block size increases, so we use ORAMs of a range of sizes, from 12 blocks (a tree of depth 1) to 4092 blocks (a tree of depth 9). The log of the time taken for each ORAM to perform 1000 iterations is plotted against the logarithm of the size of the ORAM in Figure 4.2. We expect this to be a straight line, because we expect logarithmic overheads from ORAM, and we increase the block sizes roughly in powers of two. We can see that ORAM does indeed show a logarithm overhead compared to the control experiment that performed the same sequence of accesses to a block device without ORAM. I performed the same experiment adding encryption to ORAM and we can see that the overhead was still logarithmic, but with a larger constant.

Although it has not been discussed here, bandwidth is another important measure when examining the performance of ORAM, especially when it is being used in the cloud. With more time, a detailed study of the bandwidth used with and without recursion would have been carried out, which would have given further insight into the trade-off that recursion presents us with.

## 4.3 Statistical Analysis

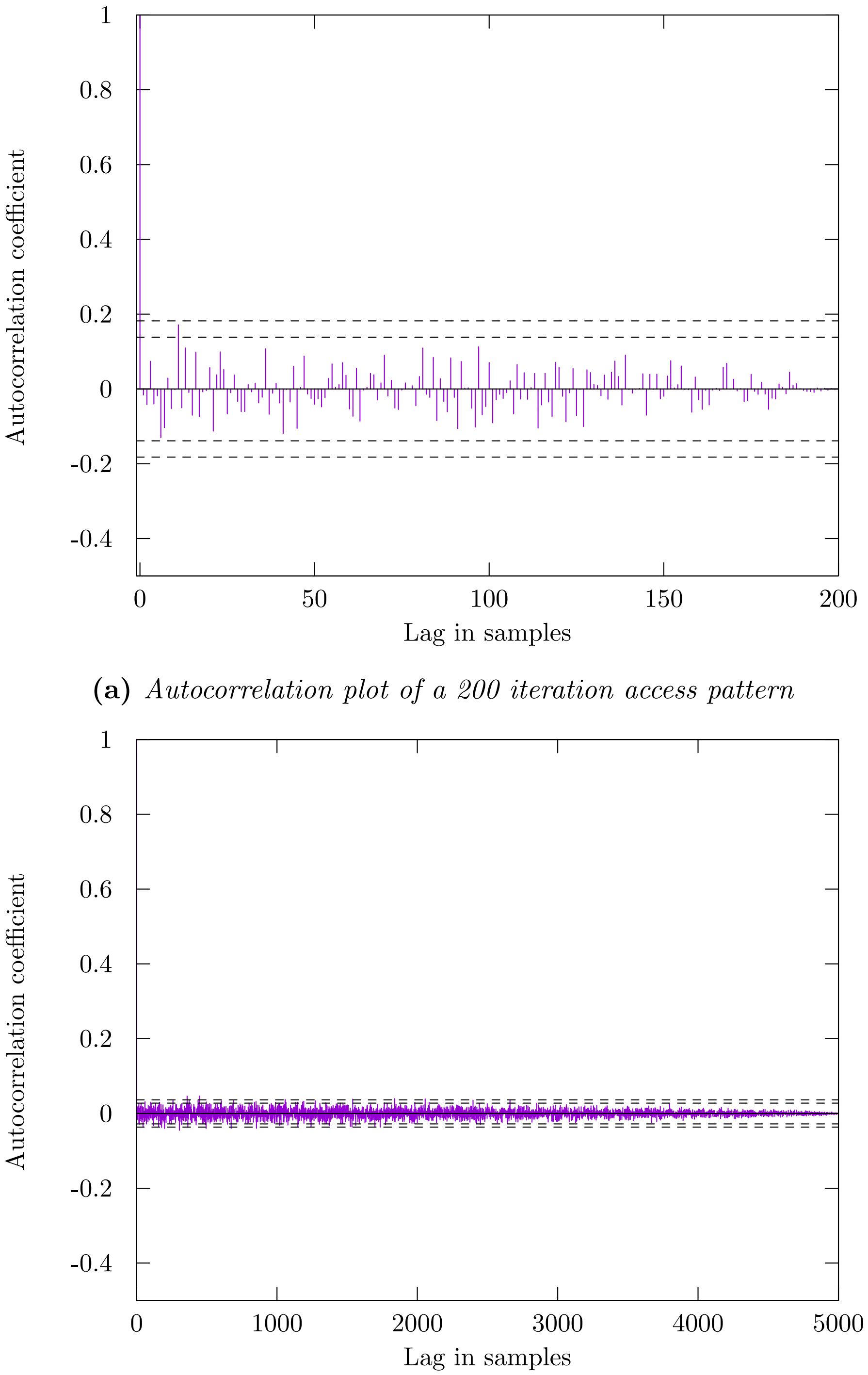
In order to test the effectiveness of ORAM, we need to show that the access pattern observed by the block device is actually statistically random. Of course there is structure to this randomness, because on each access a whole path is read and written, but the actual path that gets written should be random. Thus, we should perform a test for randomness on a sequence of path indices.

In order to do this we will use two common techniques: autocorrelation plotting and runs testing.

Autocorrelation plotting plots the correlation of a sequence with itself at various time lags [3]. For a random sequence the noise cancels itself out, so we want to see a plot of values very close to zero, apart from at lag 0, where the correlation will be exactly the power of the signal. I plotted this for two access patterns, one of length 200 so that the results would be easily visible, and one of length 5,000 to show longer term effects. For both of these patterns, the underlying access pattern, the one that would be seen without ORAM, was simply a succession of reads and writes to the same location. These plots are shown in Figure 4.3a and Figure 4.3b respectively.

The dashed black lines represent confidence bands of 95% and 99%, and in order to conclude that a sequence is random, we need almost all of the autocorrelation coefficients to lie within these bands. This is indeed the case, so ORAM has successfully taken a heavily non-random access pattern and turned it into a random one.

Runs testing attempts to detect non-random beheviour in a signal by counting the number of runs, sequences of values that are all above or below the median value. Too few runs in a sequence suggests a trend and too many runs suggests cyclic behaviour. For a large sample, we can approximate the distribution of runs using a normally distributed random variable, and compare this with the distribution that we measure. To generate the distribution, we take 1000 sample access patterns, each of length 180. We cut each pattern into 18 equally sized segments of size 10, take the mean of each segment, and count the number of runs in the sequence of means. There can be between 2 and 18 runs in each sequence, but we would expect 90% of the sequences to have between 7 and 14 runs, the 0.05% tail cut-offs for this distribution [7]. Figure 4.4 plots this distribution, along with the tail cut-offs, and 92.2% of samples fall within the bounds. Thus, we can conclude that the samples were generated from a random process.



**(a)** *Autocorrelation plot of a 200 iteration access pattern*

**(b)** *Autocorrelation plot of a 5,000 iteration access pattern*

**Figure 4.3:** *Two autocorrelation plots, with the autocorrelation coefficient on the y-axis and time lag on the x-axis. The dashed black lines represent confidence bands of 95% and 99%.*

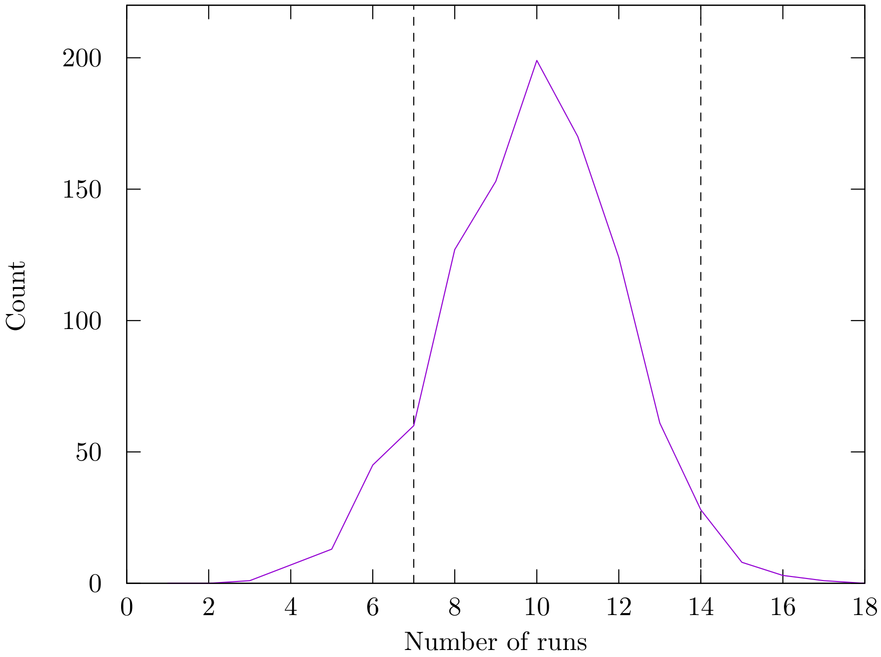


**Figure 4.4:** *The distribution of the number of runs in 1000 access patterns of length 180. The dashed black lines represent 0.05% tail cut-offs, and 92.2% of values fall within these bounds.*

# Chapter 5

# Conclusions

## 5.1 Results

I successfully implemented the Path ORAM protocol and showed that my implementation agreed with the theoretical overhead of $O(\log N)$ and that it is a statistically secure implementation.

## 5.2 Lessons Learnt

This project has taught me many important lessons. The one that has had the most impact is time management. I got stuck into the project straight away and did as much work on it as I could during the Michaelmas term and vacation, which meant that hiccoughs later down the line had a large buffer and didn't cause severe consequences. Related to this I learnt that predicting the time that each part of a project will take is very difficult and it is therefore important to be both conservative and realistic about the number of things that will go wrong. I learnt the importance of sharing your work with others through contribution to the Mirage community and being invited to talk about the project at Microsoft Research Cambridge. I would like to thank Markulf Kohlweiss from Microsoft Research for giving his advice on the project and giving me the opportunity to present my finidings. Finally, I learnt that you should seek advice as soon as there are signs of trouble, rather than waiting for a scheduled meeting to bring something up. The advice of experienced supervisors can turn the seemingly catastrophic into a minor disruption.

## 5.3 Future Work

There is still much work to be done in the field of ORAM. One of the main drawbacks of current schemes is that the size of the ORAM must be determined in advance and the overhead associated with expanding it is enormous. I would like to investigate the possibility of creating a resizable version of my implementation, however this was unfortunately not in the scope of this project.

There are also many optimisations to be performed on this implementation. Along with many interesting optimisations mentioned in the literature, there is plenty of room for analysis and optimisation of this specific implementation using benchmarking to identify code hot-spots.

I would also like to perform further analysis to get a deeper understanding of this implementation. I would like to perform further experiments on the initialisation stage of ORAM to understand how the size of the stash changes during this period. I would also like to spend more time adjusting various parameters, including block size and bucket size, attempting to find an optimal set of values for them.

# Bibliography

[1] The state of cloud storage. Technical report, Nasuni, 2013. URL http://www6.nasuni.com/rs/nasuni/images/Nasuni-White-Paper-State-of-Cloud-Storage-2013.pdf.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

[3] James J. Filliben and Alan Heckert. Autocorrelation plots. http://www.itl.nist.gov/div898/handbook/eda/section3/autocopl.htm, 2013.

[4] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[5] A. Hunt, D. Thomas, and Pragmatic Programmers (Firm). *Pragmatic Unit Testing in C# with NUnit*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2004. ISBN 9780974514024. URL https://books.google.co.uk/books?id=a61QAAAAMAAJ.

[6] 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.

[7] Maurice R. Masliah. Stationarity/non-stationarity identification. http://etclab.mie.utoronto.ca/people/moman/Stationarity/stationarity.html, 2000.

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

[9] John K Ousterhout, Herve Da Costa, David Harrison, John A Kunze, Mike Kupfer, and James G Thompson. *A trace-driven analysis of the UNIX 4.2 BSD file system*, volume 19. ACM, 1985.

[10] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[11] 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.

[12] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: practical improvements to oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, 2015.

[13] 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.

[14] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267. IEEE, 2013.

[15] 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.

[16] Paul Teeuwen. Evolution of oblivious RAM schemes. Master's thesis, Eindhoven University of Technology, 2015.

[17] Xiangyao Yu, Ling Ren, Christopher W Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Enhancing oblivious RAM performance using dynamic prefetching. *IACR Cryptology ePrint Archive*, 2014:234, 2014.

# 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 [6] 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 [15], 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 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

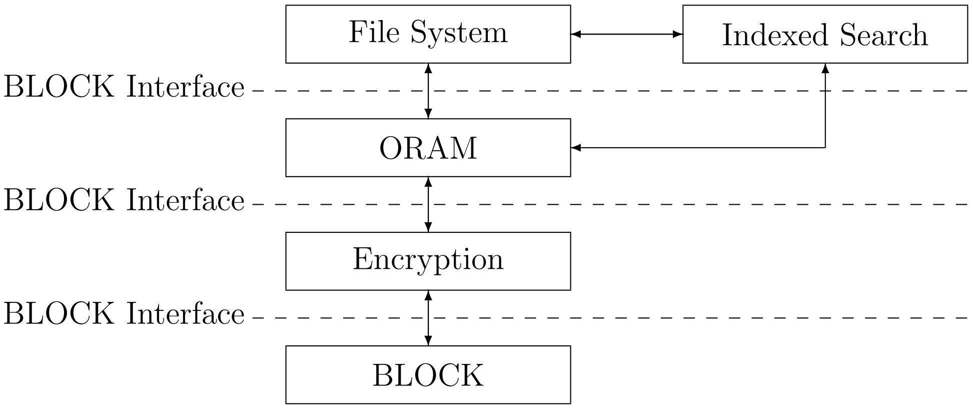


**Figure A.1:** *The application stack: ORAM satisfies MirageOS's `BLOCK` interace and any underlying `BLOCK` implementation can be used*

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 $\chi$ 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 $\chi$ 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 [11], 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.* [13] in [8], 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