Rupert Horlick

Encrypted Keyword Search Using Path ORAM on MirageOS

Computer Science Tripos – Part II

Homerton College

December 30, 2015

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

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 [1]. 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 [1] 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 [5], 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 §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 (§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 [5], 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.

1.3 Related Work

Chapter 2

Preparation

2.1 Defining the Threat Model

2.2 Introduction to ORAM

2.3 Introduction to Inverted Indexes

2.4 Requirements Analysis

2.5 Choice of Tools

2.6 Software Engineering Techniques

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 (§3.1), followed by the object store (§3.2), the index (§3.3) and finally encryption (§3.4).

3.1 Path ORAM

The structure of Path ORAM is described abstractly in [5], in terms of the core data structures

3.1.1 Stash

3.1.2 Position Map

3.1.3 Basic ORAM

3.1.4 Recursion

3.1.5 Statelessness

3.1.6 Optimisation

1. Talk about optimising the block size through design space exploration

3.2 Object Store

3.2.1 General Design

3.2.2 B-Trees

3.3 Index

3.4 Encryption

3.4.1 Library

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] 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.

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

[3] 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.

[4] 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.

[5] 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 [1] 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 [5], 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.

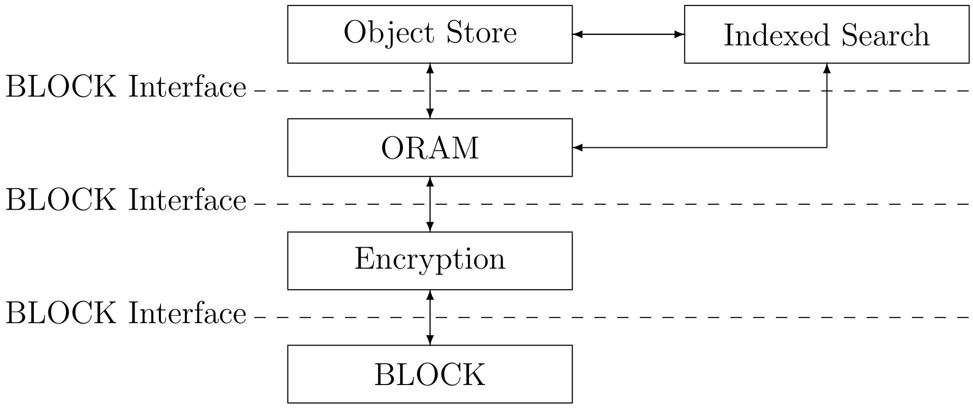

Figure A.1: The Application Stack: We can use any underlying BLOCK implementation and we can add/remove ORAM, Encryption or Search modules as we please

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 [3], 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.* [4] in [2], 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] 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.

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

[3] 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.

[4] 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.

[5] 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.