Encrypted Keyword Search Using Path ORAM on MirageOS

Rupert Horlick – rh572@cam.ac.uk

June 8, 2016

Introduction

- Final year undergraduate Computer Science student
- Undertook project over 9 months
- Implemented Path ORAM protocol, along with a file system and search module
- Evaluated performance and security properties
- Wrote 10,000 word dissertation on the whole process

Overview

Overview

Motivation

Solution

Implementation

Evaluation

Summary

Motivation

- Cloud storage's popularity demands a stronger emphasis on privacy
- Encryption hides data from cloud storage providers
  - But hinders the ability to search
- Homomorphic encryption makes encrypted search possible
  - But can leak up to 80% of queries! [Islam et al.]
- Can we have the best of both worlds?

Overview

Motivation

Solution

Implementation

Evaluation

Summary

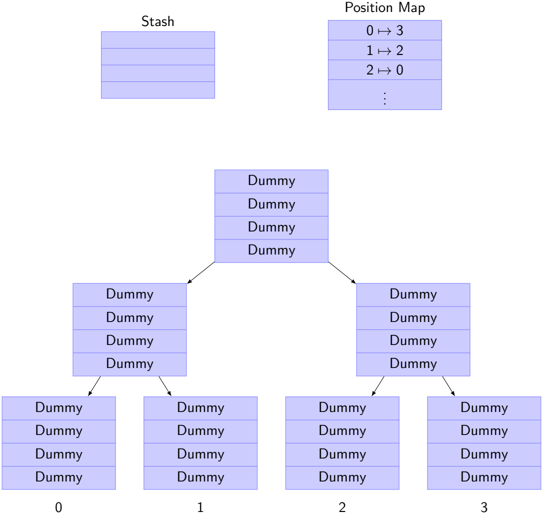
Oblivious Random Access Memory (ORAM)

- A cryptographic protocol for obfuscating access patterns
  - Trusted client and untrusted storage server
  - Relies on cryptographically secure shuffling of data
- Originally applied to software protection
  - Repurposed for secure processors and cloud computing
- Original schemes had unacceptable overheads
  - Recent improvements have made ORAM more feasible

Path ORAM

- Recent ORAM scheme (2013)
- Maintains three data structures
  - Binary tree on server
    - Each node is a bucket that contains up to Z blocks
    - Initially all blocks are dummy blocks
  - Stash on client
    - Working memory for blocks read from the tree
    - Initially empty
  - Position map on client
    - Associates to each block of data a leaf in the tree
    - Initially contains uniformly random values

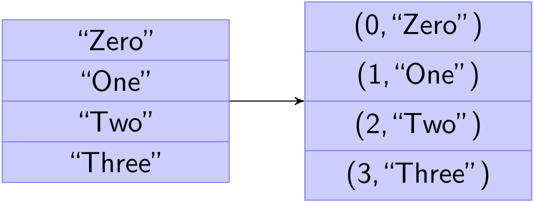
Path ORAM Initial Overview



Access Algorithm

- Signature: $\texttt{access}(\texttt{a}, \texttt{op}, \texttt{data}^*)$
- Then have the following steps:
  - Lookup position of a in position map, x
  - Remap a to a random position
  - Read the x-th path into the stash
  - If op is write, then overwrite data for a with $\texttt{data}^*$ in the stash
  - Write blocks from the stash back into x-th path
  - If op is a read, then return data

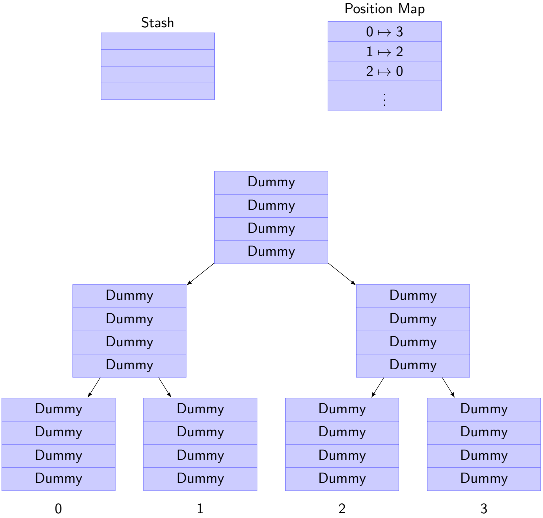
Path ORAM Input



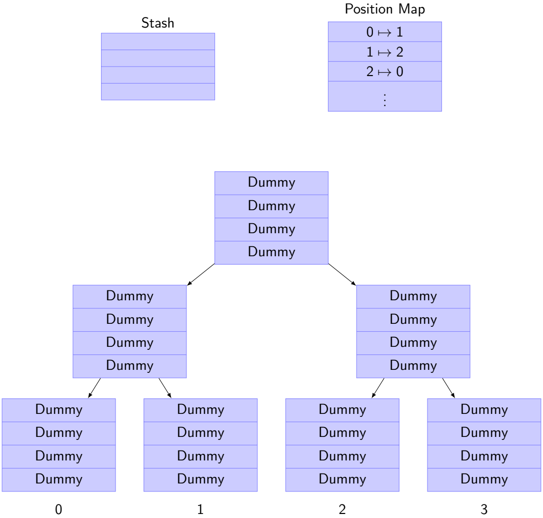
Worked Example

```
access(0,write,``Zero'')
```

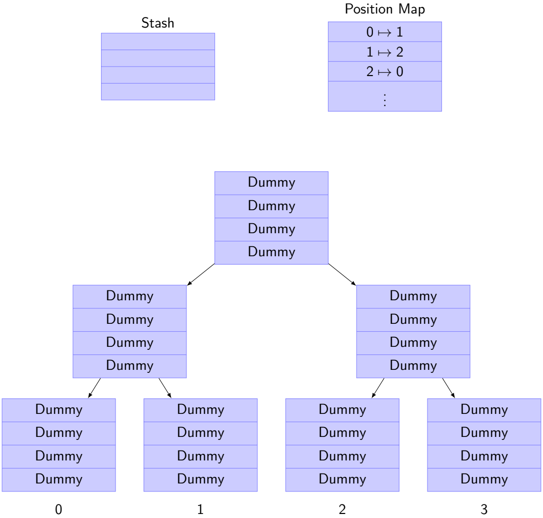
Example Write: Lookup Position



Example Write: Remap Block



Example Write: Read Path



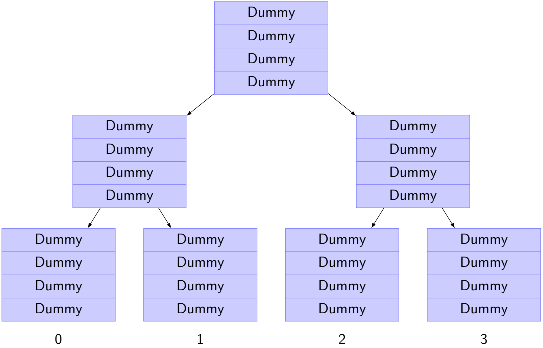
Example Write: Write Data

Stash

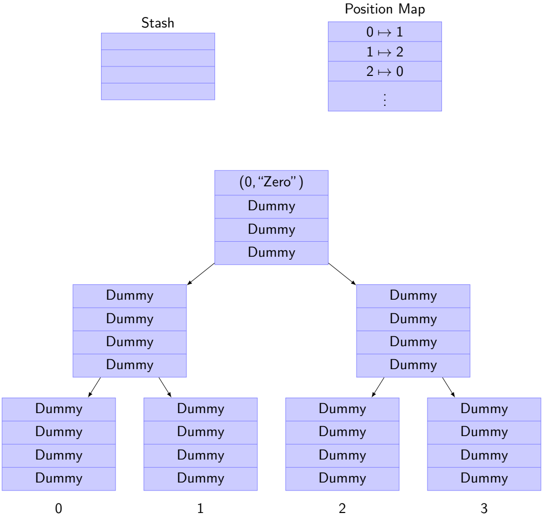
(0, "Zero")

Position Map

$0 \mapsto 1$
$1 \mapsto 2$
$2 \mapsto 0$
$\vdots$



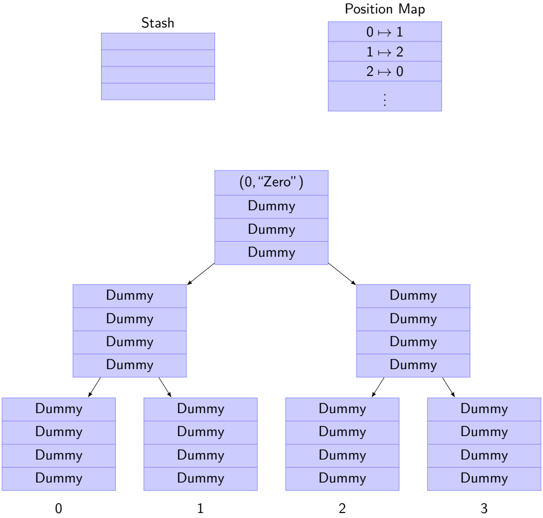
Example Write: Write Path



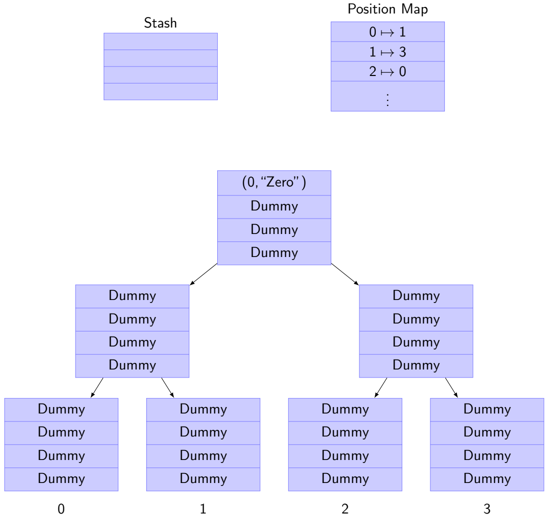
Worked Example

```
access(1,write,``One'')
```

Example Write: Lookup Position



Example Write: Remap Block



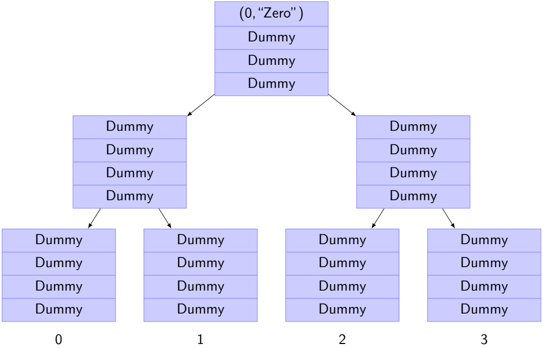
Example Write: Read Path

Stash

(0, "Zero")

Position Map

$0 \mapsto 1$
$1 \mapsto 3$
$2 \mapsto 0$
$\vdots$



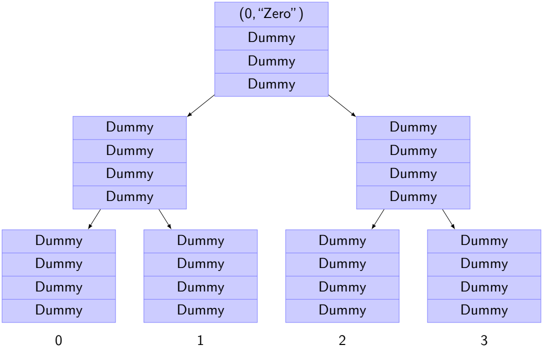
Example Write: Write Data

Stash

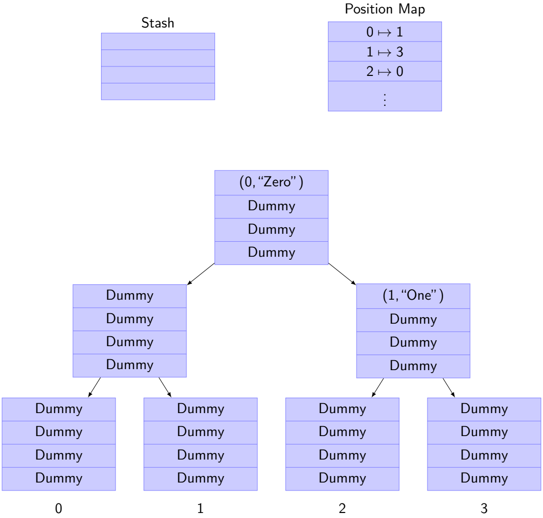
(0, "Zero")
(1, "One")

Position Map

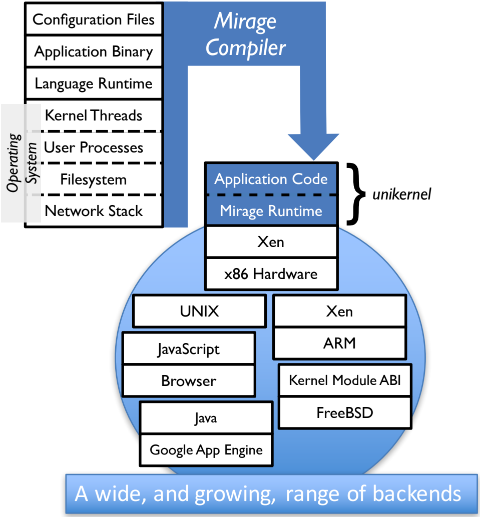
$0 \mapsto 1$
$1 \mapsto 3$
$2 \mapsto 0$
$\vdots$



Example Write: Write Path



MirageOS



Configuration Files
Application Binary
Language Runtime
Kernel Threads
User Processes
Filesystem
Network Stack
Operating System
Mirage Compiler
Application Code
Mirage Runtime
unikernel
Xen
x86 Hardware
UNIX
Xen
JavaScript
ARM
Browser
Kernel Module ABI
FreeBSD
Java
Google App Engine
A wide, and growing, range of backends

Overview

- Motivation
- Solution
- Implementation
- Evaluation
- Summary

Basic ORAM

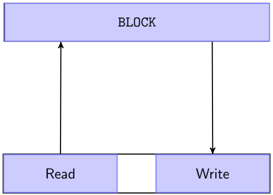
Disk

BLOCK

Basic ORAM

Encryption

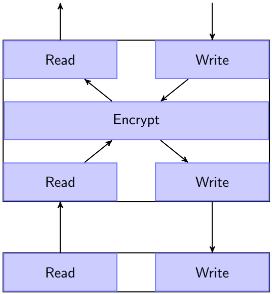
Disk



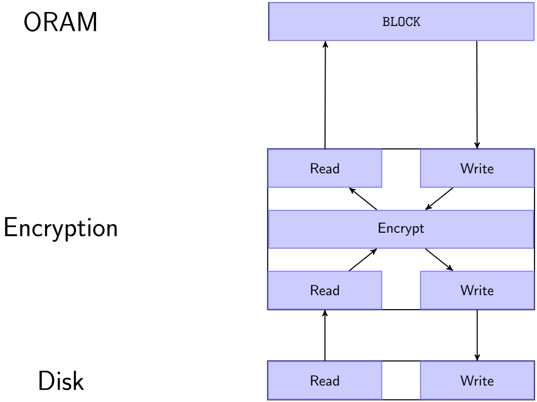
Basic ORAM

Encryption

Disk

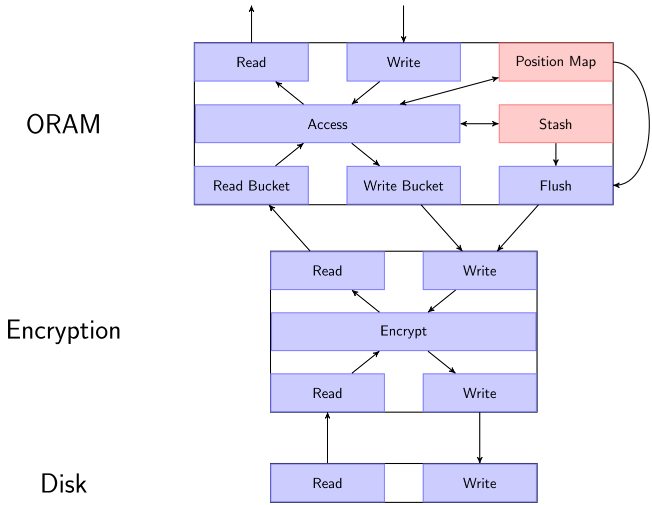


Basic ORAM



ORAM
Encryption
Disk
BLOCK
Read
Write
Encrypt
Read
Write
Read
Write

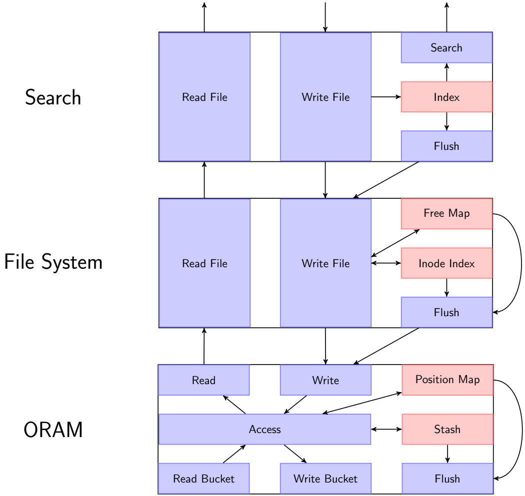
Basic ORAM


ORAM
Encryption
Disk
Read
Write
Position Map
Access
Stash
Read Bucket
Write Bucket
Flush
Read
Write
Encrypt
Read
Write
Read
Write

Recursive ORAM

- We want ORAM to be stateless, but writing position map to disk is expensive
- Recursive ORAM stores the position map of the first ORAM in another ORAM
  - The second ORAM is smaller than the first
  - This can be repeated
- Implemented this using recursive functors

Search Application



Overview

Motivation

Solution

Implementation

Evaluation

Summary

Evaluation

- Explored parameter space
  - Specifically looked at block size
  - Increasing block size increased speed
  - Chose block size of 1MB
- Measured performance
  - Compared ORAM with encryption, ORAM without encryption, and control
  - Showed expected logarithmic overheads
  - Took $\approx$1000s to transfer 1GB on 4GB ORAM
- Showed security properties using statistical techniques

Block Size Results

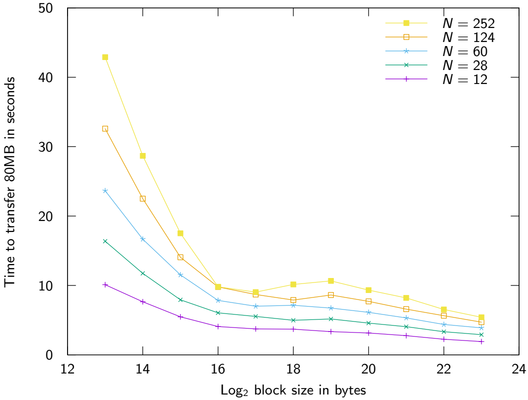


Figure: Plot of the time taken to transfer 80MB of data at varying block sizes and sizes of ORAM. Each line represents one ORAM size, N, so as block size increases, the time decreases.

Performance Results

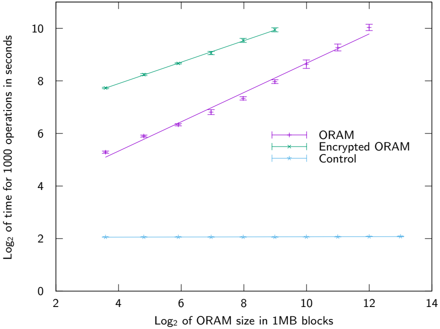
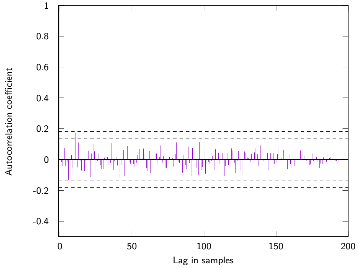


Figure: The relationship between size of an ORAM in blocks and the time taken for 1000 operations, plotted for ORAM, encrypted ORAM, and a control block device with no ORAM. We take logs of both axes, because block size was increased in powers of two and we expect a log relationship.

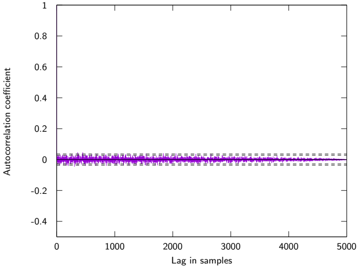
Security Evaluation

1. Autocorrelation plotting
    - Plot the correlation of a sequence with itself for a number of lags
    - For a random sequence noise cancels out to give values close to zero
2. Runs testing
    - This counts the number of runs of consecutive values all above or below the median
    - We compare this number to that of a random process

Autocorrelation Results



(a) Autocorrelation plot of a 200 iteration access pattern



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

Figure: 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%. For a random sequence, most of the points should fall within the 95% confidence bound, as they do on both of these plots.

Runs Test Results

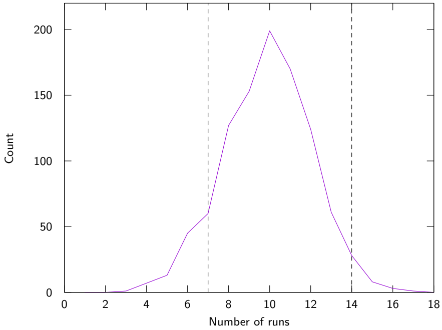


Figure: The distribution of the number of runs in 1000 access patterns of length 180. The dashed black lines represent 5% tail cut-offs. 92.2% of values fall within these bounds, implying that the access patterns were created from a random process.

Overview

Summary

- Homomorphic methods of encrypted search can leak information via the side channel of access pattern
- ORAM provides a solution to this problem
- My implementation gives the desired security properties while maintaining acceptable performance

Thank You

Questions?

https://github.com/ruhatch/mirage-oram