Encrypted Keyword Search Using Path ORAM on MirageOS

Rupert Horlick – rh572@cam.ac.uk

June 7, 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
- Wrote 10,000 word dissertation on the whole process

Overview

Overview

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!
- Can we have the best of both worlds?

Overview

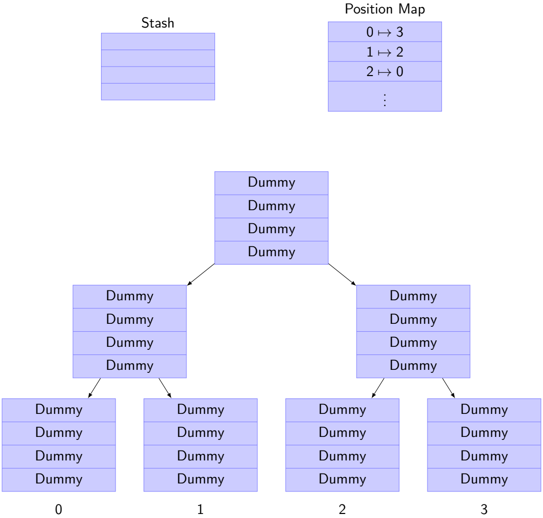
Oblivious Random Access Memory (ORAM)

- 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

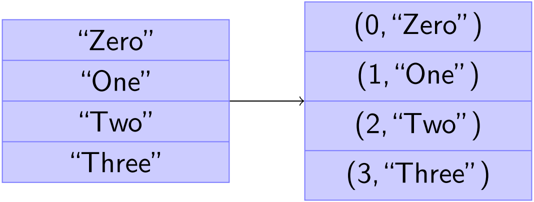
Path ORAM Initial Overview



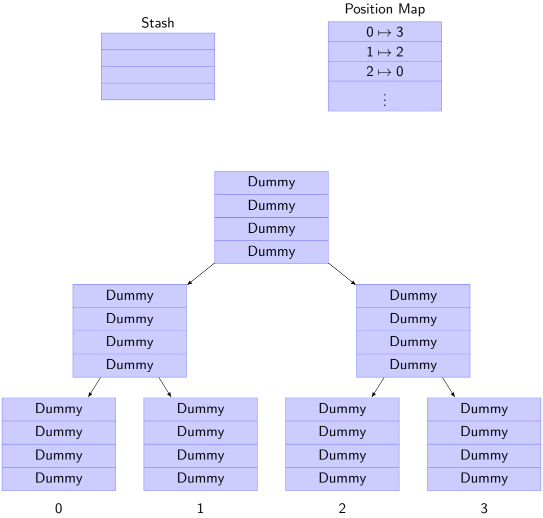
Access Algorithm

- Take in parameters a, the address, op, the operation, and $data^*$, the new data for a write
- 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 $data^*$ in the stash
  - Write blocks from the stash back into x-th path
  - If op is a read, then return data

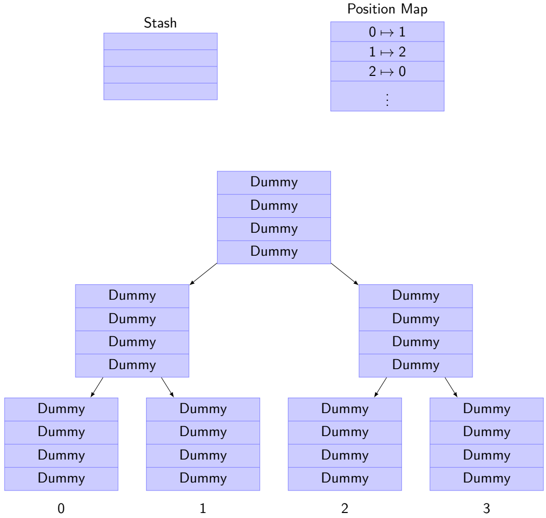
Path ORAM Input



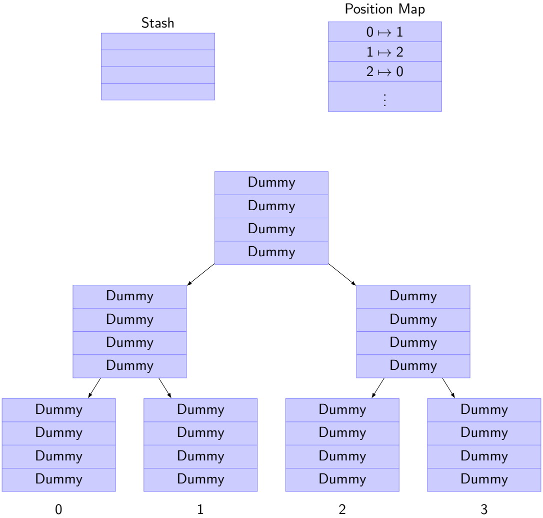
Example Write: Lookup Position



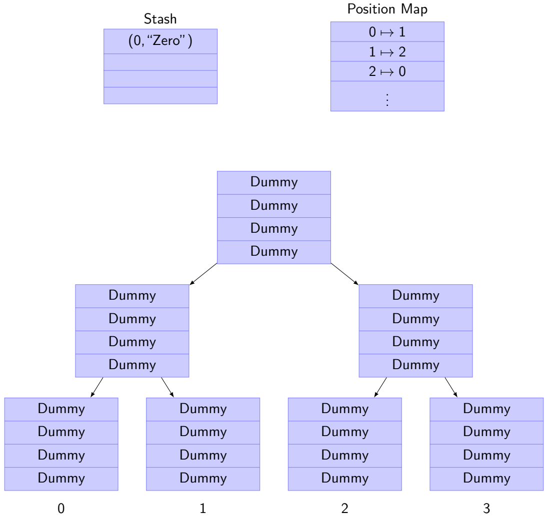
Example Write: Remap Block



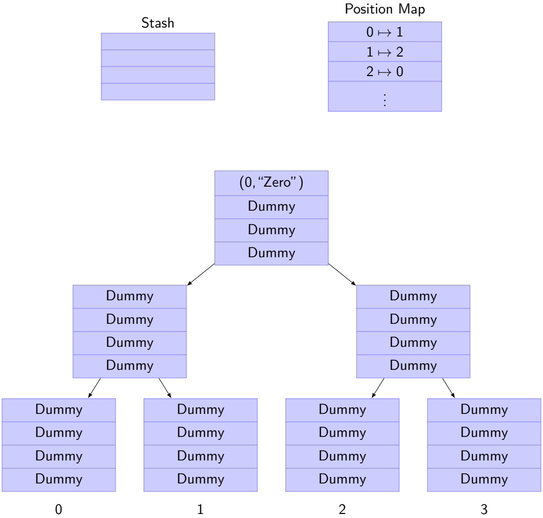
Example Write: Read Path



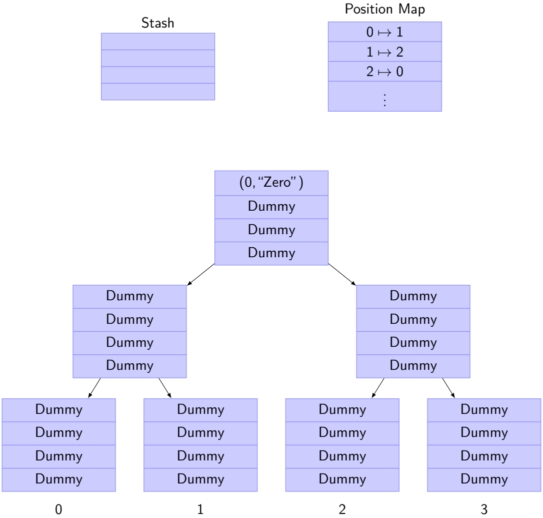
Example Write: Write Data



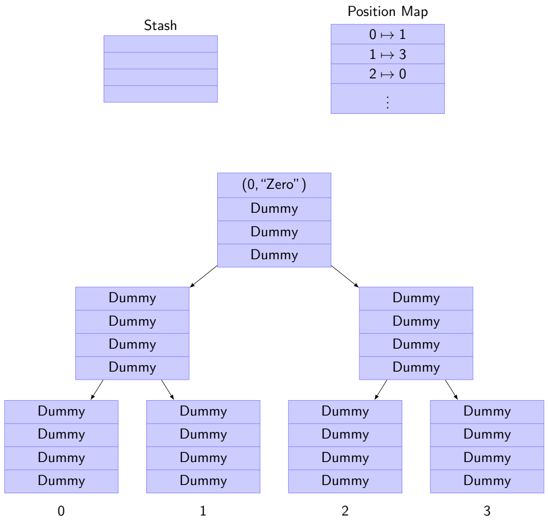
Example Write: Write Path



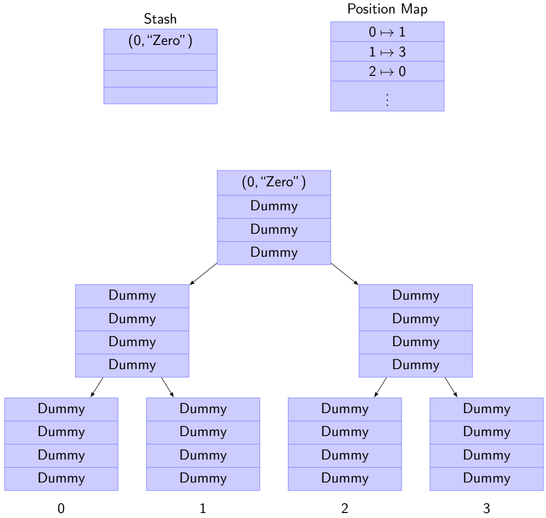
Example Write: Lookup Position



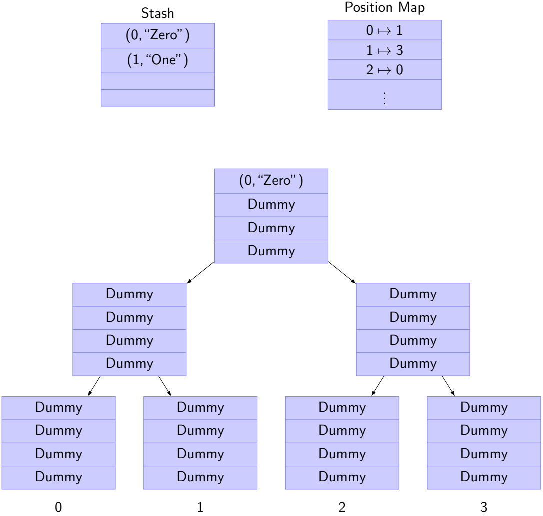
Example Write: Remap Block



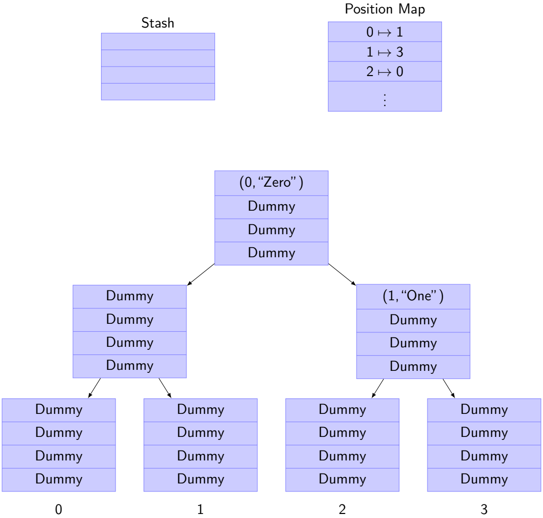
Example Write: Read Path



Example Write: Write Data



Example Write: Write Path



Introduction to MirageOS

- Library operating system, consisting of a command line tool and a set of libraries
- Compiles applications to a unikernel, a lightweight operating system
- Can be compiled for a number of targets including Xen and Unix
- Gives us control over where the ORAM application is deployed
- Allows a lightweight cloud instance to be spun up on demand

Overview

System Architecture

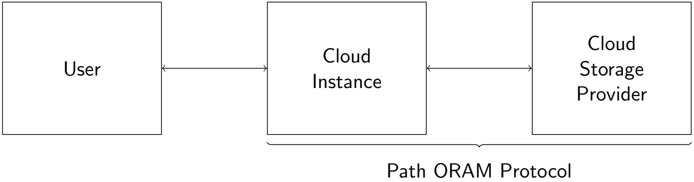


Figure: ORAM can be built as a MirageOS application, which can run on a trusted cloud instance.

Implementation of Basic ORAM

- Built Path ORAM as a functor satisfying Mirage's `BLOCK` interface
- A functor is a module parameterised over the implementation of another module
- This allows the underlying implementation of the storage device to be abstracted away
- The stash was built as a hash table
- The position map was built as a 3-dimensional array to allow 64-bit addresses

Recursive ORAM

- Do be able to disconnect from ORAM we need it to be stateless
- Writing the entire position map to disk is expensive
- Recursive ORAM stores the position map of the first ORAM in another ORAM
- This position map ORAM will have a smaller position map, because multiple addresses fit in one data block
- This process can be repeated until we have a constant sized position map
- Implemented this using recursive functors, parameterised in the position map

The Rest

- On top of ORAM I built an inode-based file system
- This included an implementation of B-Trees to store the inode index
- On top of the file system I built a simple search module
- This consists of an inverted index built on a hash table of hash sets
- I incorporated an exisiting encryption library, also built as a functor

Overview

Parameter Optimisation

- ORAM has a number of parameters that can be adjusted
- These include the size of blocks and the number of blocks in a bucket
- I focused on block size, because this has been shown to dramatically affect the speed of IO operations
- The results showed this to be true, but there is a trade off to be made
- A large block size means a larger stash, so a larger overhead for statelessness

Block Size Results

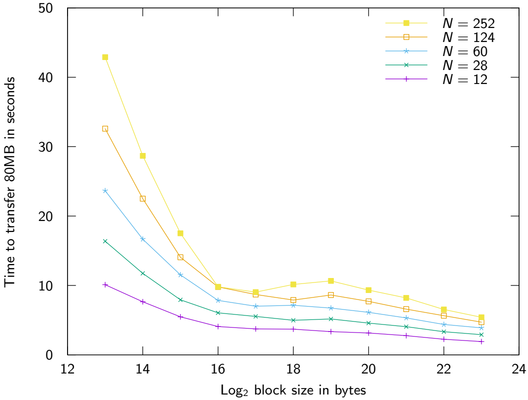


Figure: Plot of the time taken to transfer 80MB of data at varying block sizes and sizes of ORAM. Each line represents one ORAM size, N, so as block size increases, the time decreases.

Performance Evaluation

- In the Path ORAM paper, they report a theoretical overhead of $O(\log N)$
- N here is the size of the ORAM in blocks
- I performed 10 runs of 1000 iterations, for a total of $\approx$ 1GB per run
- I used ORAMs from size 12 (tree of depth 1) to 4092 (depth 9)
- The results showed a roughly logarithmic overhead, compared to the control
- Encryption added a further constant overhead

Performance Results

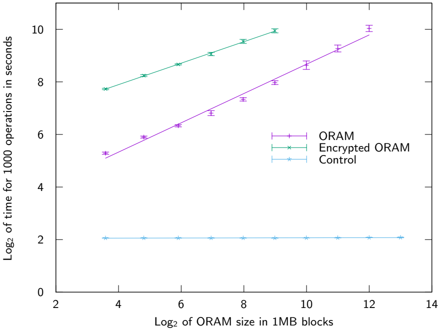
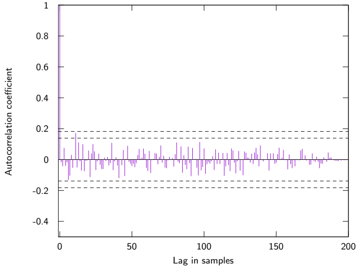


Figure: The relationship between size of an ORAM in blocks and the time taken for 1000 operations, plotted for ORAM, encrypted ORAM, and a control block device with no ORAM. We take logs of both axes, because block size was increased in powers of two and we expect a log relationship.

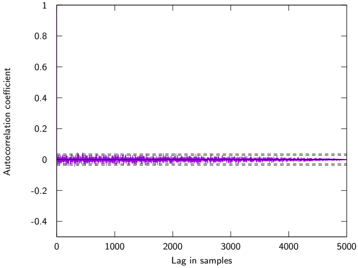
Security Evaluation

- Used statistical techniques to show that the observed access pattern is random
- The first technique is autocorrelation plotting
- This takes the correlation of a sequence with itself plotted for a number of lags
- For a random sequence, noise should cancel out to give values close to zero
- The second technique is 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

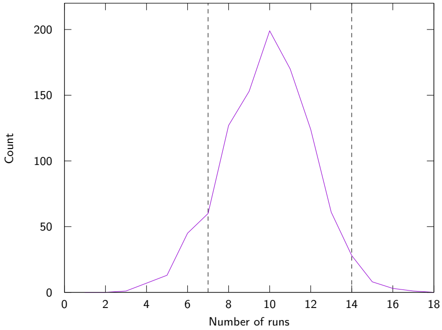


Figure: The distribution of the number of runs in 1000 access patterns of length 180. The dashed black lines represent 0.05% tail cut-offs. 92.2% of values fall within these bounds, implying that the access patterns were created from a random process.

Overview

Further Work

- Further exploration of the parameter space
- Examination of alternative performance measures including bandwidth overheads
- Deployment to the cloud and testing of overheads in real scenario

Thank you
Questions?