Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

Olav Sortland Thoresen

Oblivious RAM in practice

Master's thesis in Communication Technology
Supervisor: Colin Alexander Boyd
June 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Oblivious RAM in practice

Olav Sortland Thoresen

Master of Science in Communication Technology

Submission date:	June 2019
Supervisor:	Colin Alexander Boyd, IIK
Co-supervisor:	Gareth Thomas Davies, IIK
	Clementine Gritti, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Oblivious RAM in practice
Student:	Olav Sortland Thoresen

Problem description:

Outsourcing data storage to the cloud has become common practice both for individuals and businesses. While cloud storage offers both scalability and cost savings, it also introduces new security challenges. The cloud provider is not always trusted and sensitive data therefore needs to be protected. While encrypting the data provides some protection, it has been shown that access patterns, i.e. the order in which data is read/written, can leak information about encrypted data.

Oblivious RAM is a proposed solution to this problem. It is a mechanism to hide the access patterns of queries to outsourced data. Although the idea was proposed more than 25 years ago it was regarded mainly as a theoretical concept. New techniques have recently been proposed to make Oblivious RAM more practical, but implementations for realistic and comparable data sizes are still lacking.

This master thesis aims to make a research contribution by implementing a small number of promising candidates for Oblivious RAM, validating their constructions and assessing real running times. These experiments will consolidate and enhance understanding of the applicability of Oblivious RAM for cloud-based data outsourcing.

Responsible professor:	Colin Alexander Boyd
Supervisors:	Gareth Thomas Davies
	Clementine Gritti

Abstract

The goal of this thesis is to study the performance of Oblivious RAM (ORAM) schemes in a cloud setting and assess their practicality. To do this, three ORAM schemes (ObliviStore, CURIOUS and RingORAM) were tested on the IBM Cloud. The tests were based on five synthetic workloads and one real workload. The synthetic workloads were designed to approximate real cloud applications as closely as possible. The real workload was based on Git, a popular version control system. The results of these tests were used to compare the practicality of the three ORAM schemes for different realistic scenarios.

The results showed that ORAM still is impractical for most workloads, with costs and slowdowns being the major factors hindering practicality. Nevertheless, a handful of use-cases where current ORAM schemes can be practical, given some assumptions about the requirements for practicality, were identified. One of these scenarios is based on using ORAM to back up large (10-25 MB) email attachments. For this scenario, the slowdowns of using ORAM did not hinder its practicality and the costs were within reasonable limits for a large organization.

Apart from these findings, two other contributions were made by this thesis. An ORAM proxy was developed, allowing arbitrary cloud applications to be used with ORAM. This proxy can be used in future studies, or in practical applications. In addition, an ORAM visualizer was developed. This tool is intended for people that are unfamiliar with the concept of ORAM and provides graphical visualizations of common ORAM schemes.

Sammendrag

Målet med denne masteroppgaven er å studere ytelsen til nåværende Oblivious RAM (ORAM) løsninger og undersøke hvor praktiske de er i et skylagrings-scenario. For å gjøre dette ble tre ORAM løsninger (ObliviStore, CURIOUS og RingORAM) testet på IBMs skytjeneste (IBM Cloud). Fem syntetiske og en reell test ble gjennomført. De syntetiske testene ble utformet for å tilnærme ekte skyapplikasjoner så godt som mulig. Den reelle testen var basert på Git, et populært versjonskontrollsystem. Resultatene av disse testene ble brukt til å sammenligne hvor praktiske de tre ORAM løsningene er for forskjellige realistiske scenarier.

Resultatene av disse testene viser at ORAM fortsatt er upraktisk i de fleste tilfeller. Høye kostnader og lav hastighet er de viktigste faktorene som gjør at ORAM er upraktisk. På tross av dette ble det likvel identifisert noen scenarier der nåværende ORAM løsninger kan være praktiske, gitt noen antagelser om hva som kreves når det gjelder ytelse og kostnader. Et av disse scenariene er basert på å bruke ORAM for å sikkerhetskopiere store (10-25 MB) e-postvedlegg. For dette scenariet var nedgangen i ytelse ikke til hinder for brukbarheten og kostnadene var innenfor rimelige grenser for en stor organisasjon.

I tillegg til disse resultatene, har to andre bidrag til ORAM-miljøet blitt utvilket i løpet av denne masteroppgaven. Det første er en ORAM proxy som gjør det mulig å bruke ORAM i eksisterende skyapplikasjoner. Denne proxyen kan brukes i fremtidige studier, eller i praktiske applikasjoner. I tillegg ble det utviklet et ORAM-visualiseringsverktøy. Dette verktøyet gir grafiske visualiseringer av vanlige ORAM løsninger og er ment for å hjelpe nye personer lære om ORAM.

Preface

This thesis is being submitted in fulfillment of the final requirement of my master's degree at the Department of Information Security and Communication Technology at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisors Gareth Thomas Davies and Clementine Gritti for their guidance and invaluable feedback throughout the past year. This thesis would not have been possible without their guidance and knowledge. I would also like to thank my responsible professor Colin Alexander Boyd, both for his feedback during this thesis and for the informative and entertaining lectures he has held in his courses these past five years.

I would also like to thank the good people of room A176 for their company and support while writing this thesis. The task would have been much more arduous without their good spirits and entertaining conversations throughout the year.

Finally, I would like to thank my family for their continued support and encouragement during my studies. Special thanks to my sister Maria Sortland Thoresen for helping proof-read the thesis.

Trondheim, Friday 7th June, 2019

Olav Sortland Thoresen

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Oblivious RAM (ORAM) is a security mechanism used to hide data access patterns. The idea was proposed more than two decades ago, but it has been considered a theoretical concept for most of its lifetime. Recently, the topic has seen renewed interest with the rapid growth of cloud computing. Researchers have proposed ways to apply ORAM to cloud storage applications to provide strong, provable security. Most of this research has focused on theoretical cloud storage scenarios, with only a few papers performing real-world tests. Comparisons of proposed ORAM schemes are similarly lacking. This thesis aims to fill this gap by comparing the practicality of different ORAM schemes, in commercially available cloud storage services, using real implementations.

This chapter introduces the problem and justifies the research questions considered in this thesis. First, the problem of *secure data outsourcing* is introduced. Next, the research questions are presented and the scope of the thesis is narrowed down. Finally, an outline of the thesis is provided.

1.1 Background

Storing sensitive data in an untrusted location (e.g. a cloud server) presents a number of challenges. The data needs to be protected not only from eavesdroppers, but also from the owner of the storage itself (e.g. a cloud provider). While encryption and authentication solve some of these problems, these solutions are not always sufficient. It has been shown that *access patterns*, i.e. the order in which data is read/written, can leak information about encrypted data. An attack based on this fact was demonstrated in 2012 by Islam et al. [IKK12]. They showed that up to 80% of search queries made to an encrypted email dataset could be recovered from the generated access pattern, given some minimal background knowledge of the dataset. The aim of this thesis is to study proposed solutions to this problem (known as *Oblivious RAM* algorithms) and compare them in a real-world setting.

ORAM was initially developed by Goldreich and Ostrovsky [Gol87, Ost90, GO96] as a way to protect software from reverse-engineering. They proposed a solution where software would be sold in a package containing a secure processor and an encrypted version of the software. The decryption key would be embedded in the secure processor and physical shielding would be used to protect it. This software/hardware package would then be connected to the customer's computer, allowing it to use the memory and other available peripherals (keyboard, mouse, etc.). To prevent the customer from reverse-engineering the software, an ORAM algorithm would be employed. The ORAM algorithm would act as an intermediary between the software and the memory, obscuring the true access pattern of the software.

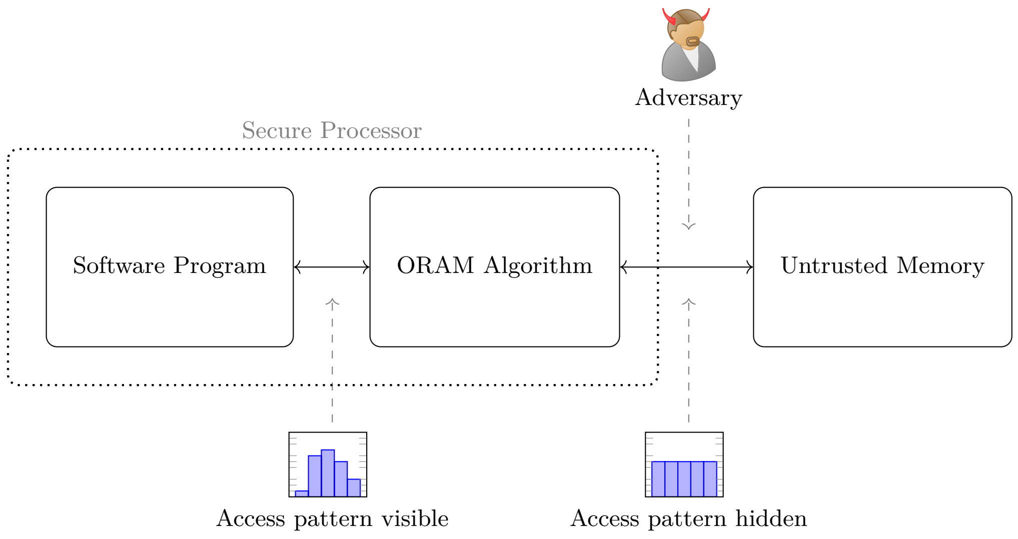


Figure 1.1: The classical ORAM model.

A model of this scenario is shown graphically in Figure 1.1. The figure shows the secure processor running a program. The program needs to interface with untrusted memory in a secure way. To achieve this an ORAM algorithm sits between the program and memory. The algorithm takes requests from the program and turns them into a string of seemingly random requests to the memory. This obscures the true access pattern from an adversary observing the traffic to/from the memory.

In recent years, a branch of ORAM research has focused on applications in cloud storage. Outsourcing sensitive information to the cloud presents challenges similar to those faced in the software protection case. However, instead of protecting software from reverse-engineering, the focus is on protecting data stored in the cloud from the service provider that hosts it. This is known as the secure data outsourcing problem.

Figure 1.2 illustrates the problem graphically. A client (e.g a smartphone, a desktop computer, etc.) communicates with an untrusted cloud. The link between the client and the cloud is encrypted and secure against eavesdroppers. However, the client's access pattern can still be observed by an adversary that has access to the cloud storage. This could be the owner of the cloud or an adversary that has gained similar privileges (e.g through a security breach). To protect against this, the client uses ORAM to hide its access pattern. This scenario is known as the *public-cloud scenario*.

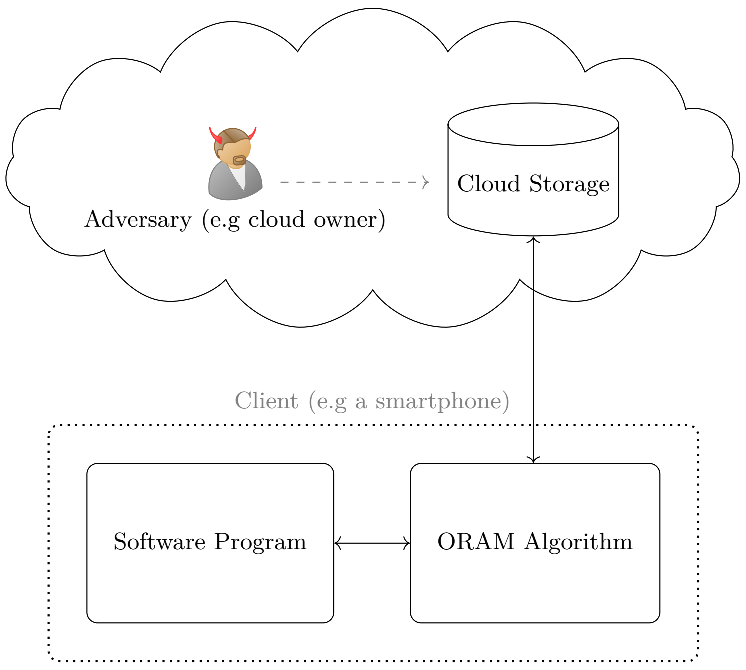


Figure 1.2: Secure Data Outsourcing - The public-cloud ORAM model.

Using ORAM in this way provides the desired level of security, but adds substantial bandwidth overheads to the link between the client and the cloud. If this link has limited capacity (e.g when using a cellular network) these overheads can be prohibitively high. An optimization that can be made is to move the ORAM algorithm away from the client. This is shown in Figure 1.3. In this scenario, the client instead communicates with a server on a private cloud. The private cloud is trusted and the client therefore does not need to hide its access pattern. The server is responsible for the communication with the untrusted cloud storage and thus employs ORAM. In this scenario, the bandwidth intensive ORAM operations occur between the two clouds, relaxing the requirements on the (usually more constrained) link to the client. This scenario is known as the *private-cloud scenario*.

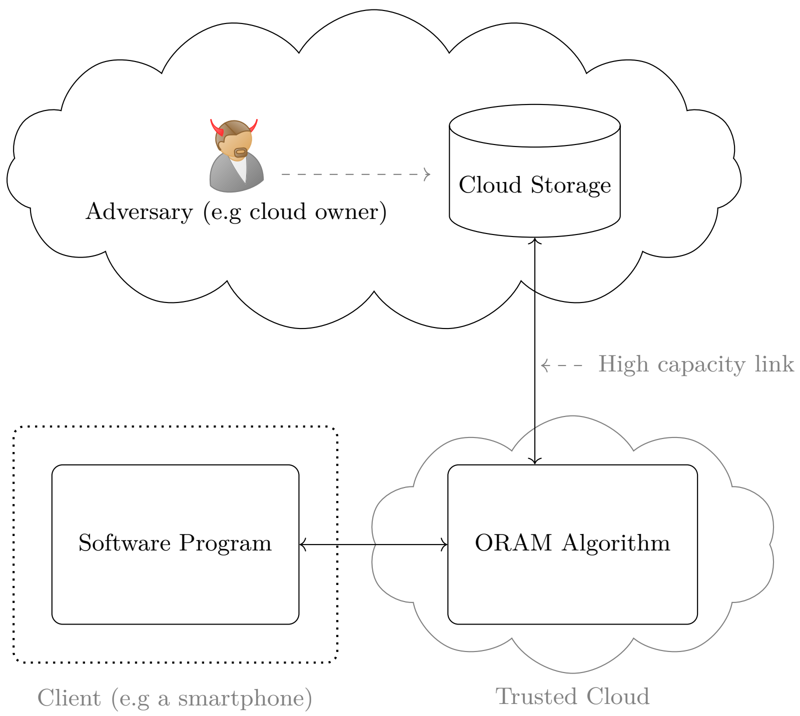


Figure 1.3: Secure Data Outsourcing - The private-cloud ORAM model.

Existing ORAM schemes can be used in both of these scenarios, however, a study by Bindschaedler et al. [BNP$^+$15] showed that traditional ORAM schemes performed poorly in a cloud setting. The study also identified some new metrics (response time, monetary expenses, etc.) that should be considered when evaluating ORAM schemes for use on the cloud. These metrics are important for practical performance, but have traditionally not been considered in ORAM research.

1.2 Justification

As mentioned in the previous section, Bindschaedler et al. [BNP$^+$15] studied the performance of a selection of ORAM schemes in a cloud setting. To do this, they ran tests on Amazon S3, both with and without ORAM, for a small set of workloads. They claimed that these workloads were realistic and that their results thus reflected the usability of ORAM at the time. This study is, to the best of the authors knowledge, the only systematic study of ORAM performance in a cloud setting.

While their study is an important step on the way to practical ORAM, it has some weaknesses that raise questions about how applicable their results are to real-world applications. For instance, the workloads were generated by mounting an Amazon S3 bucket as a folder, running the benchmarking software *Filebench*[1] on this folder and recording the requests received by Amazon S3. This is problematic because Filebench is designed to benchmark traditional storage devices, like hard drives or solid state drives. Cloud storage solutions are typically optimized for other usage scenarios. Dewan et al. [DH11] underline this fact, in a survey of cloud storage services, by stating that: "cloud storage [...] is not exactly the same as that of a local file system. Hence, integration with a local file system requires manipulations at several levels right from mimicking the recursive directory namespace to the control of the access control list." The developers of real-world cloud applications are likely to be aware of this and their applications will therefore be designed around the strengths and limitations of cloud storage. It is therefore likely that a gap between the results of Bindschaedler et al. [BNP$^+$15] and the real-world exists.

Another problem with the study lies in the last two steps of their methodology; collecting a log of requests from Amazon S3 and replaying this log through an ORAM scheme. The log contains the most important information about the requests made to the S3 service, i.e. which types of requests were performed, how many bytes were uploaded/download, timestamps, etc. However, it does not contain any information about which requests were dependent on previous requests.

To illustrate why this is a problem, an example can be used. Imagine that an application is reading a linked list stored in the cloud. The elements of the list are stored in random locations on the cloud server (due to additions/deletions, other data, etc.). Thus, to read an element in the list, all previous elements must be accessed. This means that the application will first send a request for the first element, wait for the response from the cloud, then request the next element, etc., until the desired element is found. This process cannot be paralellized, but under the methodology used by Bindschaedler et al. it is not possible to distinguish it from a sequence of independent, parallelizable requests. This means that a parallel ORAM scheme would perform unrealistically well when tested with this sequence, compared to if the ORAM scheme was implemented as a part of the application.

The study by Bindschaedler et al. [BNP$^+$15] was conducted in 2015. A substantial amount of research has since been conducted and new ORAM schemes have been proposed. Because of this, and the problems mentioned above, more work in the area of practical ORAM is required.

[1]Filebench, https://github.com/filebench/filebench

1.3 Research Questions

The aim of this thesis is therefore to answer the question:

– Are current ORAM schemes practical in a cloud setting?

To answer this question, the following sub-questions will need to be answered:

– Which metrics determine the practicality of ORAM in a cloud setting?
– Which types of workloads are found in practical cloud applications?
– How do current ORAM schemes perform under realistic workloads?
– What are the costs associated with using ORAM with cloud storage?
– Are there other factors that need to be considered when using ORAM in practice?

This will be done by repeating the study of Bindschaedler et al. [BNP^{+}15], but with a new methodology and implementation. New ORAM schemes such as RingORAM (see Section 3.8.2) will also be included.

1.4 Scope

A wide variety of ORAM schemes have been proposed in recent years. These schemes are often tailored towards specific application scenarios. Because of this great diversity, it is necessary to narrow down the scope of the thesis and focus on a small selection of schemes. Since this thesis focuses on practical applications, only schemes that can be implemented on current cloud storage solutions are considered. In addition, to ensure that results are comparable, only schemes that adhere to the mainstream definition of ORAM (see Section 3.1) are considered. The schemes that have been selected are presented in Section 4.1.

Some ORAM schemes require server-side computations. These requirements range in complexity from simple XOR-operations to full-blown general computation. The APIs of common cloud storage services do not currently offer this functionality. Because of this, schemes requiring server-side computations will not be considered in this thesis.

Other types of schemes that are out of scope for this thesis are Oblivious Paralell RAMs (schemes designed to support parallel accesses, see Section 3.12) and Differentially Private ORAMs (schemes that do not guarantee perfect privacy, instead providing a tradeoff between privacy and performance, see Section 3.11). These schemes deviate from the mainstream definition of ORAM and will therefore not be considered.

In this thesis, only the two public-cloud and private-cloud scenarios mentioned in Section 1.1 will be considered. In addition, because of time restrictions, only the scenario where the client is a desktop computer, with ample local storage and a reliable, high-capacity link to the cloud, will be considered.

1.5 Thesis Outline

The thesis is organized into 7 chapters as follows:

- **Chapter 1:** Introduction and motivation for the thesis.
- **Chapter 2:** Important concepts and terminology that will be used throughout the rest of the thesis.
- **Chapter 3:** An introduction to the concept of ORAM, a key theoretical concept in the thesis.
- **Chapter 4:** Description of the methodology used in the thesis, including the test setup and the workloads that were selected. This chapter also describes the components that had to be implemented and some challenges that were faced during implementation.
- **Chapter 5:** A summary of the results that were gathered. Interesting and/or unexpected results are highlighted and commented on in this chapter.
- **Chapter 6:** Discussion of the results and their implications in the context of the research questions.
- **Chapter 7:** Conclusion of the thesis, and proposed future work.

Chapter 2

Preliminaries

This chapter presents some important concepts and definitions that will be used throughout the rest of the thesis. First, the trend of data outsourcing is discussed, followed by a short comparison of cloud storage services, a section on asymptotic notation and a note on units. Note that the concept of Oblivious RAM (ORAM) is not covered here, as it has been dedicated its own chapter (Chapter 3).

2.1 Data Outsourcing

The volume of data created and stored, by both businesses and individuals, has increased rapidly over the last decade. This trend is expected to continue, with IDC estimating that the total amount of data created, captured or replicated will grow from 33 Zettabytes[1] in 2018, to 175 Zettabytes in 2025 [RGR18]. Keeping all this data backed up, secured and available is becoming increasingly challenging. While businesses are faced with increased management costs [Avr14], individuals are struggling with the increasing need for technical know-how [GL18].

Because of this, many are choosing to outsource their data to a third party. This moves the burden of storing the data from the owner to the third party. In most cases, the third-party is a *cloud storage provider* that provides storage services to a large number of users. This can yield additional benefits from the point of view of the data owner [Köh15]. A key benefit is *scalability*, since the cloud provider maintains a large pool of resources ready to handle increases in data volume. Another benefit is the *cost savings* incurred by not having to invest in hardware upgrades. Most cloud providers operate with a pay-as-you-go business model, meaning that the data owner only has to pay for the resources it actually uses. Finally, the *technical know-how* of operating a high-availabilty and high-capacity storage service has to be maintained by the cloud provider. This reduces the need for IT-staff for businesses and makes the service more accessible to individuals.

[1]A zettabyte is approximately one trillion gigabytes, or 2^{70} bytes

2.2 Cloud Storage

The adoption of cloud storage (and other cloud services) has grown rapidly in recent years. Accompanying this growth has been a substantial amount of "hype" which has caused some confusion regarding the precise definition of a cloud service. This section aims to address this by giving the definitions that will be used throughout the thesis.

The National Institute of Standards and Technology (NIST), an American standards agency, has created a definition of cloud computing [MG11]. The definition is general enough to be useful, but at the same time specific enough to capture the nuances of the term as it is used in this thesis. Whenever the terms *cloud computing* or *cloud service* are used, they refer to the NIST definition, as stated below:

Definition 2.1. NIST definition of cloud computing [MG11]
Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Cloud storage is the use of cloud computing to provide storage services. The service providers take advantage of the economic benefits of pooling storage resources to offer users a cost-effective and convenient way to store arbitrary amounts of data.

Current cloud storage services are quite diverse when compared to traditional storage solutions. Storage media like HDDs and CD-drives use standard connectors and interfaces (e.g, SATA and SAS), making it easy to switch to a different storage medium when the need arises. The same is not true for cloud storage services which often require the use of proprietary Application Programming Interfaces (APIs) to interact with the service. Services also differ in the way they logically structure the storage. Some services support a traditional file hierarchy (e.g personal cloud storage services like Dropbox or Google Drive) while others treat the storage as a collection of objects (e.g Amazon S3, OpenStack Swift). The latter are referred to as *object storage* services. Three object storage services, that are central in this thesis, are covered in more detail in Sections 2.2.1 to 2.2.3.

2.2.1 Amazon S3

Amazon Simple Storage Service (Amazon S3) is a prominent example of a cloud storage service. It was introduced in 2006 and was one of the first publicly available services adopting the object storage paradigm. Amazon S3 organizes the storage into *buckets*, where each bucket can hold an arbitrary amount of objects. Each object has a unique identifier and holds a variable amount of data. In addition, objects can store custom metadata that can be used in searches.

Amazon provides an API[2] that can be used to interact with the storage. Every call to this API constitutes a simple operation, like downloading an object or creating a bucket. Each operation has an associated cost, with upload operations being more expensive than download operations. When calculating a users bill, Amazon tallies up the cost of all operations performed during the billing period. Additional charges are added for the amount of bandwidth and storage used.

2.2.2 OpenStack Swift

OpenStack Swift (Swift) is a popular alternative to Amazon S3. It is a part of the OpenStack ecosystem, a suite of open source solutions for cloud services originally developed by Rackspace and NASA [Lam14]. Swift, like Amazon S3, is an object storage service. Objects are organized into *containers*, the Swift equivalent of buckets. Containers, like buckets, can not be nested meaning that only a one-level hierarchy is supported.

Swift has an API[3] that provides functionality similar to that of the Amazon S3 API. Since Swift is an open source project no fixed pricing scheme exists, instead prices are decided by the cloud providers that deploy it. Examples of cloud providers that deploy Swift are Dell, IBM and Rackspace.

2.2.3 IBM Cloud Object Storage

The IBM Cloud was chosen as the representative cloud service for this thesis (see Section 4.4). Because of this, the object storage service available in this cloud (IBM Cloud Object Storage) was used. This service is compatible with Amazon S3 and uses the same logical structure (buckets and objects).The Amazon S3 API is used to access the service, ensuring compatibility with existing cloud applications.

2.3 Git

Git was used in some of the tests that were conducted. A primer on Git terminology is therefore included here. For a full introduction to Git, the reader is referred to the book *Pro Git* by Scott Chacon and Ben Straub [CS14].

The most important term in Git is the *repository.* Git repositories can be thought of as folders with version control. When multiple people are collaborating on a project using Git, they are all making changes to the same repository. A consequence

[2]Amazon S3 API, https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
[3]OpenStack Swift API, https://developer.openstack.org/api-ref/object-store/

of this is that their local repositories may be in different states. This is resolved by a user *pushing* their changes to a central server, called the Git server. When new changes are pushed, they are *merged* into the repository on the server. To receive these changes, a different user has to *pull* the changes from the server. When the changes are pulled, they are merged into the local repository of the user. In Git, changes are referred to as *commits*.

The first time a user wants to access a repository, he/she has to *clone* the repository from the server. This operation downloads the entire repository, including the *commit history* (i.e the history of changes) to the users computer. After this operation is performed, the users repository is in sync with that of the server, until any new commits are pushed.

2.4 Asymptotic Notation

Asymptotic notation (also known as Big-O notation) is a mathematical notation used to describe how a function grows when its argument approaches infinity. The notation is often used in computer science to compare the running time of algorithms. In ORAM literature it is used when describing the theoretical overheads of ORAM schemes. The notation is occasionally used in this thesis and a short introduction is therefore given in this section.

Symbol	Meaning	Example
O	Upper bound	$3N+1$ is $O(N^2)$ (i.e grows no faster than N^2)
Θ	Tight bound	$3N+1$ is $\Theta(N)$ (i.e grows with the same rate as N)
Ω	Lower bound	$3N+1$ is $\Omega(\log N)$ (i.e grows no slower than $\log N$)

Table 2.1: The three symbols used in asymptotic notation and their meanings.

In mathematics, asymptotic notation is used to describe the growth of functions. This is useful when comparing functions that have long, complicated expressions. Consider the two functions $f(N) = 2^N + 5N^3 + 15N^2 + N\log N + 1$ and $g(N) = 2N^3 + 5N^2 + 15N + \log N + 1$. At first glance these functions might look similar, but upon closer inspection one can observe that one function grows much faster than the other. Asymptotic notation can be used to efficiently express these types of relationships between functions.

Three different symbols are used in asymptotic notation. These are shown in Table 2.1. The first symbol, O (Big-O), is used to give an upper bound on the growth of a function. Using the two functions from the last paragraph as an example, we can say that both f and g are $O(2^N)$. This means that both functions grow slower or at the same rate that 2^N grows. This is true since the factors dominating the growth of the two functions are 2^N for f and N^3 for g. Both of these factors are smaller than or equal to 2^N. Note that we ignore the constant factors (e.g for g we ignore the 2 in $2N^3$) when discussing growth, since these factors become negligible as N approaches infinity. We can also say that g is $O(N^3)$, but f is not.

The two remaining symbols are used in a similar fashion. The symbol Ω (Big-Omega) is used to give lower bounds, while Θ (Big-Theta) is a shorthand for saying that a function has the same upper and lower bound. For example, we can say that f is $\Theta(2^N)$. This means that f is both $O(2^N)$ and $\Omega(2^N)$.

Asymptotic notation is often used in ORAM papers to describe the bandwidth overhead incurred by an ORAM scheme when a block is read or written. This overhead typically depends on the number of blocks in the ORAM, commonly referred to as N. A common overhead is $O(\log N)$, meaning that at most $\log N$ blocks need to be accessed for each read or write operation. A summary of the theoretical overheads of the ORAM schemes considered in this thesis is given in Table 3.1.

A weakness of using asymptotic notation when describing ORAM schemes is that it hides constants that could potentially be significant. An ORAM scheme with a bandwidth overhead of $2 \log N$ appears similar to a scheme with an overhead of $2000 \log N$ when asymptotic notation is used. Because of this, it is not possible to determine the practicality of an ORAM scheme from its theoretical overhead only.

2.5 Note on Units

There are differing opinions in the IT industry regarding the uses of prefixes when discussing bytes. For example, some people think that kilobytes (KB) should refer to $2^{10} = 1024$ bytes while others prefer $10^3 = 1000$ bytes. Throught this thesis, the traditional units are used, meaning that KB refers to $2^{10} = 1024$ bytes, MB refers to $2^{20} = 1048576$ bytes, and so on. The reasoning behind this choice is that the majority of the previous works that are referred to throughout the thesis uses these units.

Chapter 3 Oblivious RAM

This chapter gives an introduction to the concept of ORAM, a key theoretical concept in this thesis. First, a definition is given, followed by an overview of the history as well as descriptions of a selection of ORAM schemes. For readers unfamiliar with ORAM, the visualization tool that was created while researching for this thesis might be useful while reading this chapter. It is described in Section 6.4 and a live version is available at `http://folk.ntnu.no/olavsth/oram`.

3.1 Definition

Informally, ORAMs are mechanisms for secure data outsourcing. They allow data to be stored in an untrusted location without leaking any information about the data to the owner of the storage. This is achieved in two steps. First, the data is encrypted before it is sent to the remote storage. This protects the data both in transit and when sitting "idle" in the remote storage (assuming a secure and properly implemented encryption scheme is used). Encryption does not however, protect the *way* the data is accessed. When the data is read or written to by the user, an *access pattern* is generated. The access pattern can potentially reveal a lot of information about the data, depending on the nature of the data and how it is accessed.

A more formal definition of ORAM is given by Ren et al. [RFK+15]. In their definition the untrusted storage is referred to as the *server*, being accessed by a *client*. Note that slightly different terminology is used in this thesis. The term *server* is only used when referring to a cloud server, while *untrusted storage* or *external storage* is used in the general case. The definition is restated (with some slight changes to notation) in Definition 3.1.

Definition 3.1. (ORAM Definition)
Let

$$\vec{y} = ((\mathrm{op}_M, \mathrm{addr}_M, \mathrm{data}_M), ..., (\mathrm{op}_1, \mathrm{addr}_1, \mathrm{data}_1))$$

denote a data sequence of length M, where op_i denotes whether the i-th operation is a read or a write, addr_i denotes the address for that access and data_i denotes the data (if a write). Let ORAM($\vec{y}$) be the resulting sequence of operations between the client and server under an ORAM algorithm. The ORAM protocol guarantees that for any $\vec{y}$ and $\vec{z}$, ORAM($\vec{y}$) and ORAM($\vec{z}$) are computationally indistinguishable if $|\vec{y}| = |\vec{z}|$, and also that for any $\vec{y}$ the data returned to the client by ORAM is consistent with $\vec{y}$ (*i.e., the ORAM behaves like a valid RAM*) with overwhelming probability.

3.2 History

The study of ORAMs started with the works of Goldreich and Ostrovsky [Gol87, Ost90] in the 1980s. Their research was motivated by the growing trend of software piracy. They argued that existing anti-piracy solutions both lacked sufficient theoretical treatment and were vulnerable to reverse-engineering. Both of these issues were addressed in a 1996 paper [GO96] that proposed ORAM as a potential solution. The paper proposed some ORAM schemes, studied their performance and laid the theoretical groundwork for future ORAM research.

The best-performing scheme proposed by Goldreich and Ostrovsky was based on a hierarchical construction. This construction was used in subsequent research [WS08, PR10, KHK10] to create ORAM schemes with incrementally better performance. These schemes can be classified as *Hierarchical ORAMs* (sometimes called *Layered ORAMs*).

Other types of ORAMs have since been developed. Shi et al. [SCSL11] were the first to drop the hierarchical construction and propose an ORAM scheme based on a binary tree structure. By doing this, they achieved better practical performance compared to other schemes. Their scheme, along with several improved versions [SvDS$^+$13, RFK$^+$15] can be classified as *Tree-Based ORAMs.* These schemes are discussed in Section 3.8.

Stefanov et al. [SSS11] suggested a new way to design ORAM schemes, based on partitioning. They suggested dividing the remote storage into smaller ORAMs, called Sub-ORAMs. This design allowed multiple requests to be processed at the same time if the requested blocks reside in different Sub-ORAMs. Prominent examples of this design are ObliviStore [SS13] and CURIOUS [BNP$^+$15]. These schemes are known as *Partition-Based ORAMs* and are covered in detail in Section 3.10.

Goodrich et al. [GMOT12] proposed a solution based on caching. In their solution, blocks are cached by the client when they are first accessed. All subsequent read or write operations only affect the local copy, ensuring that the access pattern is hidden from the server. When the cache eventually fills up, the remote storage is shuffled and the changes from the cache are written back in an oblivious way. Their scheme and its derivatives are called *Large-Message ORAMs* and are discussed in Section 3.9.

A new ORAM category was recently introduced by Wagh et al. [WCM16]. They relax the guarantees traditionally provided by ORAM, allowing a trade-off between privacy and security. They call this type of ORAM a *Differentially Private ORAM* (DP-ORAM). Under the traditional definition, an ORAM scheme guarantees that any two access patterns are indistinguishable from the point of view of the external storage. A Differentially Private ORAM instead gives some probabilistic guarantees of how a small change in the input to the ORAM will affect the access pattern seen by the external storage. These schemes are covered in Section 3.11, but are not further considered in this thesis.

A branch of ORAM research has focused on making ORAMs parallel. These schemes, while out of scope for this thesis, are briefly discussed in Section 3.12.

3.3 Applications

Although ORAMs were originally developed as a solution to software piracy, applications of the technology have been found in many different fields, both theoretical and practical. At the time of writing these applications are mostly within academia.

The design of secure processors is perhaps the application that is closest to what Goldreich and Ostrovsky originally envisoned. ORAMs have proven to be useful in this area, with both Maas et al. [MLS$^+$13] and Ren et al. [RFK$^+$18] opting to use PathORAM [SvDS$^+$13] in their implementations of secure processors.

Secure Multi-Party Computation (MPC) is another research area where applications for ORAM have been found. The goal of MPC is to allow "a group of mutually distrustful parties to compute a joint function on their inputs without revealing any information beyond the result of the computation" [HHNZ19]. One way to achieve this is to use ORAMs to build secure computation protocols. An example of this is SCORAM by Wang et al. [WHC$^+$14].

This thesis focuses on applications of ORAM in cloud storage. As mentioned in Chapter 1, outsourcing sensitive information to the cloud presents challenges similar to those faced in the software protection case. However, instead of protecting software from reverse-engineering, the focus is on protecting data stored in the cloud from the service provider that hosts it. Bindschaedler et al. [BNP$^+$15] published a paper on this topic where they studied the practical performance of a selection of ORAM schemes in a cloud setting. They identified some metrics (response time, monetary expenses, etc.) which are important for practical performance, but have traditionally not been considered in ORAM research. Their paper is an important source for this thesis.

3.4 Building blocks

This section covers some of the building blocks that are commonly found in ORAM schemes. These will be referenced in sections 3.5 - 3.12 where the schemes are explained in detail.

3.4.1 Data Blocks

An ORAM consists of a (usually large) number of data blocks stored in some untrusted location. To maintain obliviousness, all blocks are of the same fixed size. Some ORAM schemes, such as the *Square-Root ORAM* described in Section 3.6, require additional blocks that are not used for client data. This means that an ORAM of size N blocks, as seen by the client, could require $N + M$ blocks of remote storage to function. In this case M can be both dependent and independent of N, depending on which ORAM scheme is used.

It should not be possible for an adversary to differentiate between used and unused blocks. To ensure this, dummy blocks are used. These blocks contain a special value to indicate that they are empty. Since blocks are encrypted with a probabilistic encryption scheme (see Section 3.4.3) only the client can distinguish the dummy blocks from real blocks.

3.4.2 Position Map

When ORAM is used, blocks in the remote storage are shuffled randomly. This means that a given block might not reside in the same location it would reside in if ORAM was not in use. Because of this, two different *address spaces* are often used when discussing ORAMs. The *virtual* address space refers to the addresses seen by the client, while the *physical* address space refers to the addresses that the blocks actually reside in at a given time. To keep track of the mapping between these two address spaces, a *position map* is used. When the client asks for a block, the ORAM scheme looks up its virtual address in the position map and finds the corresponding physical address of the block. This is shown in Figure 3.1. The ORAM scheme ensures that the position map is always up to date whenever blocks are moved or shuffled in the remote storage.

The size of the position map is dependent on the size of the ORAM. This can be a problem if size of the clients trusted storage is small. To alleviate this problem and

Virtual Address	1	2	3	4	5	...	N
Physical Address	7	5	8	2	4	...	1

Figure 3.1: Example of a position map. The mapping between the virtual and physical address of block #2 is highlighted.

achieve a constant client side storage independent of the ORAM size, recursion can be used.

3.4.3 Encryption

Hiding the access pattern is only useful if the data itself is also hidden. Otherwise, the adversary can recover the access pattern simply by tracking data blocks by their content as they move around. Blocks therefore have to be encrypted before they are sent to the remote storage.

Special care must be taken when selecting an encryption scheme. An adversary should not learn anything by inspecting and comparing the encrypted blocks. This means that two identical blocks must not look the same after encryption. Re-encrypting a block without changing its contents must also produce a different encrypted block, so that it is not possible to tell if blocks where changed. This can be achieved using a *probabilistic* encryption scheme, meaning that randomness is used during encryption.

Since encryption is required in all ORAM schemes, it is often not mentioned explicitly in the literature. This is also the case in this thesis.

3.4.4 Stash/Shelter

Some ORAM schemes reserve a special area for short-term storage of blocks. In Tree-Based ORAM schemes, this area is located in the trusted client storage and is referred to as the *stash*. The Square-Root ORAM and its derivatives place this area in the untrusted remote storage and refer to it as the *shelter*.

3.4.5 Oblivious Scan

Searching through an array to find an element is a common task in many algorithms. This is traditionally done using a *linear scan*, which starts from one side of the array and accesses every element until the desired element is found. If the array is sorted a *binary search* can be used, which improves performance by continually halving the search space, inspecting only the middle element and excluding one side from the search. Both of these approaches give the desired result, but they are non-oblivious.

This means that their access pattern depends on the input (in this case the array to search and the desired element). An adversary can thus ascertain information about the input from the access pattern. In the case of a linear scan, it is easy to see that this is true. Assuming that the desired element was found, it was found in the last location accessed before the scan terminated. This holds true for the binary search as well. In addition, the binary search reveals information about how the desired element compares to the other elements in the array, since the array is sorted.

The traditional linear scan and binary search can therefore not be used in ORAM schemes. An oblivious version of the linear scan can be created by making two simple changes to the algorithm: 1. Always access all elements (do not stop after desired element has been found). 2. Whenever an element is accessed it should both be read from and written to. This oblivious linear scan is used in many ORAM schemes, most notably the Trivial ORAM scheme described in section 3.5. It is also possible to create an oblivious version of the binary search (as demonstrated by Gentry et al. [GGH$^+$13a]) but this is complicated and not commonly used as a building block in ORAM schemes. It is therefore not further elaborated here.

Adopting the notation used in [Tee15], the keyword **scan** denotes an oblivious scan when used in pseuodocode.

3.4.6 Oblivious Sort

Sorting is another common operation that requires special care when obliviousness is required. Most ordinary sorting algorithms, such as Quick Sort or Merge Sort, are non-oblivious, meaning that their access pattern depends on the contents of the array being sorted. Because of this, they are not suitable for use in ORAM schemes.

There exists sorting algorithms where the access pattern is independent of the input. These are known as oblivious sorting algorithms and are used in some ORAM schemes, particularly in Goldreich and Ostrovsky's hierarchical construction and its derivatives (for more details, see section 3.7)

One way to achieve oblivious sorting is to use a *sorting network*. Sorting networks have a fixed number of inputs and outputs, as well as a fixed number of *comparison elements*. These elements have two inputs (A & B) and two outputs (MIN & MAX). Their function is to send the smallest of the inputs to the MIN output and the largest to the MAX output. By connecting many such comparison elements together, networks capable of sorting a large number of elements can be constructed. Since these networks are fixed, the comparisions they make are not dependent on the array being sorted. Thus they are oblivious. An example of such a network is Batcher's Sorting Network [Bat68]. This was the network used in the original hierarchical ORAM scheme proposed by Goldreich and Ostrovsky [GO96].

Sorting networks are not the only way to construct oblivious sorting algorithms. Randomized versions of classic sorting algorithms can also be oblivious. Goodrich [Goo10] demonstrated this by creating a randomized version of shellsort. While these algorithms usually outperform sorting networks, they can not guarantee that the output will be fully sorted due to their randomized nature. Although the failure probability is small, it still needs to be considered in practical implementations. Williams et al. [WST12] tried using the randomized shell sort in their ORAM scheme, but ran into problems. In their paper, they remarked that: "when applied to ORAM, the sort parameter k must be chosen to guarantee success with overwhelming probability. [...] it is unclear what parameters can be chosen for a practical implementation."

Adopting the notation used in [Tee15], the keyword **sort** denotes an oblivious sort when used in pseuodocode.

3.5 Trivial ORAM

A very simple (but inefficient) ORAM scheme can be constructed from the observation that the access pattern generated from accessing every block in the remote memory does not yield any useful information for an attacker. It is therefore sufficient to ensure that every block is both read and written to whenever a block is requested by the client. This is known as the *Trivial ORAM* scheme, first proposed in Goldreich and Ostrovsky's original paper [GO96]. It is described in pseudocode in Algorithm 1.

Algorithm 1: Trivial ORAM (Adapted from [Tee15])

```
function ACCESS(op, addr, value):
    scan blocks from 0 to $N$:
        if $block_i$ has virtual address $addr$ then
            if op = read then
                $tmp \leftarrow block_i$
            else
                $block_i \leftarrow value$
    if op = read then
        return tmp
```

Although this algorithm is very simple, it scales very badly. An ORAM of size N requires N read operations and N write operations, for a total of $2N$ operations every time a block is accessed. This quickly becomes too expensive as the size of the ORAM grows. Asymptotically, the access overhead of the Trivial ORAM is $O(N)$ which is far worse than alternative schemes, which usually have an access overhead of $O(\text{poly} \log n)$. (where "$\text{poly} \log n$" means "some polynomial in $\log n$")

3.6 Square-Root ORAM

Goldreich and Ostrovsky [GO96] proposed the Square-Root ORAM as the first step towards an efficient ORAM scheme. The main idea is to avoid having to access $O(N)$ blocks and instead only access $O(\sqrt{N})$. To achieve this, a special layout is used. The remote storage is divided into two areas, the *permuted memory* and the *shelter*. The permuted memory consists of the N original data blocks as well as $\sqrt{N}$ dummy blocks. This area is called permuted because it is regularly shuffled while the ORAM is in use. The shelter can hold up to $\sqrt{N}$ blocks and is reserved for short term storage. The full layout is shown in Figure 3.2.

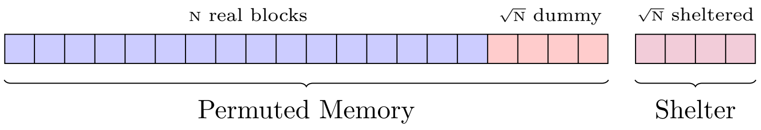


Figure 3.2: Layout of the remote storage in the Square-Root ORAM.

The execution of the Square-Root ORAM is divided into *epochs*. An epoch starts with the permuted memory being shuffled randomly. Next, a total of $\sqrt{N}$ virtual accesses are handled. When handling a virtual access, the entire shelter is first scanned. If the requested block is not found in the shelter, it is read from its current location in the permuted memory. Otherwise a dummy block is read from the permuted memory. In any case, either the original value (in case of a read) or the new value (in case of a write) is then written (obliviously) to the shelter. Once $\sqrt{N}$ virtual accesses have been handled, the blocks in the shelter are written (obliviously) back to the permuted memory and a new epoch starts.

The Square-Root ORAM is described in pseudocode below. For a more detailed explanation, including proofs, the reader is referered to the original paper [GO96].

3.7 Hierarchical Schemes

Goldreich and Ostrovsky [GO96] showed that the theoretical lower bound for ORAM schemes is $\Omega(\log N)$. To get closer to this bound they proposed a new scheme that achieved a poly-logarithmic overhead. They achieved this by extending the Square-Root ORAM. The basic idea is to view the Square-Root ORAM as having two levels; the first being the stash and the second the permuted memory. This construction can be extended to form a hierarchy of levels, where every level acts as stash for the next. By making each level twice as large as the previous, a total of $\log N$ levels are required.

Algorithm 2: Square-Root ORAM (Adapted from [Tee15])

```
function ACCESS($op, addr, value$):
    if $t = 0$ then
        BEGIN_EPOCH()
    else if $t = \sqrt(N)$ then
        END_EPOCH()
    else
        scan blocks in the shelter:
            if $block_i$ has virtual address $addr$ then
                $tmp \leftarrow block_i$
                $InShelter \leftarrow true$
        if $InShelter$ then
            $dummy \leftarrow block_{\pi(N+t)}$                  ▷ Reads a dummy block
        else
            $tmp \leftarrow block_{\pi(addr)}$
        if op = write then
            $tmp \leftarrow value$
        scan blocks in the shelter:
            write a block with address $addr$ and data $tmp$ into available slot
    $t \leftarrow t + 1$
    if op = read then
        return tmp

function BEGIN_EPOCH():
    $\pi \leftarrow$ a random permutiation of $[0, N + \sqrt{N}]$
    scan blocks in the permuted memory:
        tag $block_i$ with $\pi(i)$
    sort blocks on the tagged value
    return $\pi$

function END_EPOCH():
    scan blocks in the shelter:
        tag $block_i$ as "new"
    sort entire memory on virtual addresses
    scan entire memory:
        if $block_i$ and $block_{i+1}$ have the same virtual address then
            overwrite the oldest block with a dummy block
```

In the Square-Root ORAM, the entire shelter is obliviously scanned every time it is accessed. Doing this in the hierarchical solution would be too expensive, as every level would have to be scanned. Because of this, levels are treated as hash tables instead of arrays. This way, the location of a block can be looked up before it is accessed. While this improves performance, it raises some new problems. The first problem is that an adversary might be able to predict which location a block will map to in the hash table, depending on the hash function used. If a well known hash function is used, the adversary can pre-compute the location for a large number of virtual addresses and create a lookup table. To address this problem, the Hierarchical ORAM uses a secret, random, hash function for every level in the hierarchy.

The second problem is that hash tables are prone to collisions, meaning that two different blocks could end up in the same location. To solve this problem, entries in the hash table are treated as *buckets*. Each bucket can hold a fixed number of blocks. Collisions are therefore not a problem unless buckets become full. The probability of this happening is small if each bucket can hold $O(\log N)$ blocks and the hash function distributes blocks uniformly [GO96]. The full layout, with levels, buckets and blocks is shown graphically in Figure 3.3.

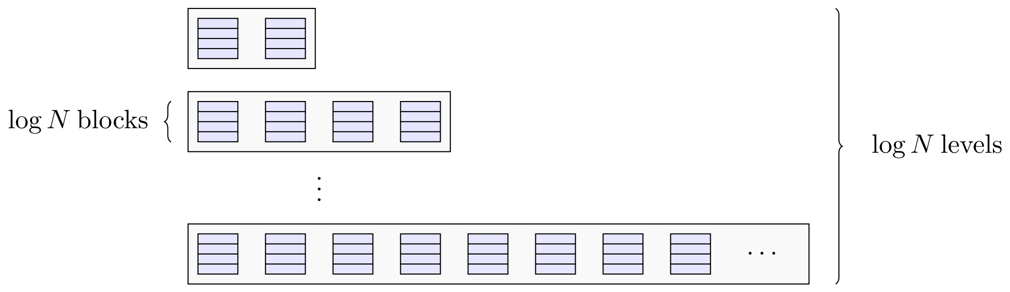


Figure 3.3: Layout of the remote storage in Hierarchical ORAM schemes.

To access a specific block, referred to as $block_i$, the following steps are taken:

– for every level, until $block_i$ has been found:
  ◦ look up the bucket corresponding to $block_i$ is in the hash table
  ◦ scan the bucket
  ◦ if $block_i$ is found, store it in a temporary variable

– read dummy blocks from all remaining levels

– if the operation is a write, update the temporary variable

– write the temporary variable to the highest level of the hierarchy

The advantage of this hierarchical structure is that it allows for longer epochs. The biggest overhead of the Square-Root ORAM is the reshuffling operations that happen at the beginning/end of an epoch.

3.8 Tree-Based Schemes

Tree-Based ORAMs organize the remote storage in a binary tree structure. Each node in the tree is a bucket that can hold a fixed number of data blocks. Every block is assigned to a random leaf node and will at any time reside in one of the buckets on the path to this leaf node. Whenever a block is accessed, it is assigned to a new leaf node and placed at the top of the tree. Blocks are periodically moved down the tree in a process called *eviction*. This design was first proposed by Shi et al. [SCSL11] and has since been improved upon in several papers [GGH$^+$13b, CP13, SvDS$^+$13, CLP14, WCS15, RFK$^+$15]. An illustration of the design is shown in Figure 3.4.

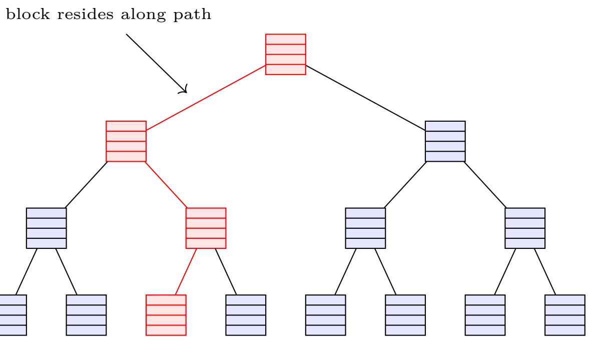


Figure 3.4: Layout of the remote storage in Tree-Based ORAM schemes.

The main innovation of Tree-Based ORAMs is that these schemes do not require periodic reshuffling of the remote storage (an expensive operation). Instead the shuffling is performed in smaller steps, with every eviction step performing a partial shuffling operation. While the main goal of the eviction is to move blocks down the tree, a side effect is that blocks are moved towards their randomly assigned positions. Over time the remote storage will become completely shuffled, compared to its original state.

A drawback of this approach is that the eviction process can fail. As the ORAM fills up, more and more blocks will be assigned to the same path in the tree, eventually causing the fixed size buckets to become full (this is often referred to as an *overflow* in the literature). When this happens, blocks can no longer be evicted from the root node and it is also filled up. This causes the ORAM scheme to enter a failure state from which it cannot recover. The original paper [SCSL11] addressed this issue by arguing that the failure probability is small, but this is only true in their theoretical analysis. For practical implementations with restrictions on local and remote storage, this becomes a real issue. Because of this, subsequent papers have addressed this problem more thoroughly.

Asymptotically, Tree-Based ORAMs have the same overhead as Hierarchical ORAMs, however simulations have shown that they perform better under practical workloads [SCSL11].

3.8.1 PathORAM

A notable Tree-Based ORAM scheme is *PathORAM* [SvDS$^+$13], due to its ability to give a theoretical bound on the local memory usage. This is useful in constrained devices like embedded systems or mobile phones.

Path ORAM extends the original binary tree construction by introducing a *stash*. The stash is similar to the shelter from the Square-Root ORAM in that it is used to temporarily store blocks. It can only store a fixed number of blocks, independent of the size of the ORAM (unlike the shelter) and it can be located in either local or remote storage. The stash is used both as a cache, to improve performance, and as a place to store overflowing blocks. This reduces the probability of ending up in the failure state described in the previous section.

For a full description of PathORAM, including proofs and practical experiments, see the original paper by Stefanov et al. [SvDS$^+$13].

3.8.2 RingORAM

Ren et al. [RFK$^+$15] proposed an improved version of PathORAM. Their scheme, known as *RingORAM* achieves 2.3× to 4× better bandwidth and allows tuning the size of the local memory for different application scenarios. These improvements makes RingORAM the most interesting Tree-Based ORAM scheme for practical applications.

These improvements are achieved using several new techniques and insights. The bandwidth requirements are reduced by making the bandwidth independent of the bucket size. In PathORAM, reading a block from a bucket amounts to reading and writing all the blocks in that bucket to maintain obliviousness. In RingORAM, only one block needs to be read. For data blocks, this is still oblivious since they are immediately assigned to a new path in the tree after being read. This is not true for dummy blocks, so some periodic reshuffling of buckets are required. If a bucket has S dummy blocks, it will need to be reshuffled after it has been accessed S times.

Additionally, RingORAM uses a better eviction strategy. This strategy is based on performing evictions on paths in a lexicographical order, instead of evicting on the most recently accessed path. This spreads eviction paths more evenly over the tree. Because of this, evictions can happen less frequently while still maintaining a low failure probability.

For a full description of the scheme, see the paper by Ren et al. [RFK$^+$15].

3.9 Large-Message Schemes

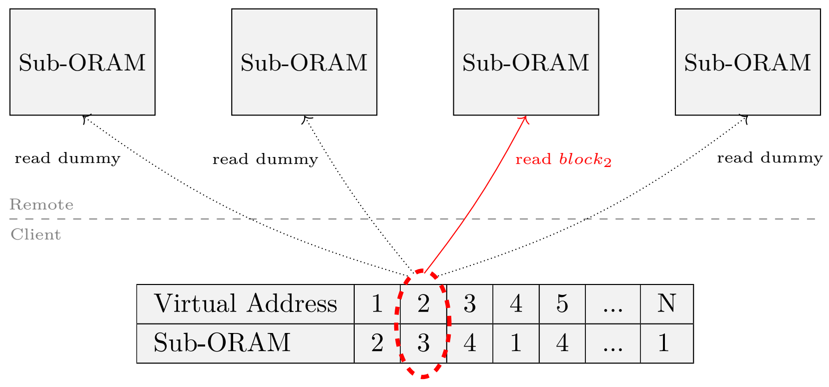
Large-Message ORAMs rely heavily on caching. Whenever a data block is fetched from the external storage, it is cached by the client. If the same data block is requested later, it will be fetched from the cache without any interaction with the remote storage. This ensures that blocks are only accessed once. When the cache is full, the remote storage has to be reshuffled and the blocks in the cache are written back obliviously.

A drawback of this approach is that updated blocks are only written back to the remote storage once the cache fills up. If the client crashes, all changes since the last reshuffle are lost.

An example of a Large-Message ORAM scheme is PracticalOS by Goodrich et al. [GMOT12].

3.10 Partition-Based Schemes

Partition-Based ORAMs divide the external storage into a set of smaller partitions. Each partition is a fully functioning ORAM (often refered to as a *partition ORAM* or *Sub-ORAM*) and the client uses a position map to keep track of which partition a data block resides in. When blocks are retrieved, they are stored for a random amount of time in the client's *eviction cache* before being written back to the external storage. This design is notable because it supports concurrent requests if the requested blocks reside in different partitions.



Virtual Address	1	2	3	4	5	...	N
Sub-ORAM	2	3	4	1	4	...	1

Figure 3.5: Accessing a block in a Partition-Based ORAM scheme.

When accessing a block in a partition based scheme, the following steps are carried out (these steps are shown graphically in Figure 3.5):

– look up the virtual address of the block in the position map to find the corresponding partition.

– forward the block request to the Sub-ORAM in that partition.

– request a dummy block from the Sub-ORAMs of all other partitions.

– assign the block to a new random partition and add it to the eviction cache.

The third step is crucial for maintaining obliviousness. Without this step, an adversary would learn which partition a given block resides in, effectively revealing the position map stored in the clients trusted memory.

The blocks in the eviction cache have to be written back to the remote storage at some point. This task is handled by the eviction process, which runs as a background process. Different schemes use different eviction strategies, but a possible strategy is *random eviction*. This strategy was described by the authors of the partition based scheme ObliviStore [SS13] as follows: "with every data access, randomly select 2 partitions for eviction. If there exists a block in the eviction cache that is assigned to the chosen partition, evict a real block; otherwise, evict a dummy block to prevent information leakage." Evicting blocks at random times in this way ensures that an adversary can not correlate a block access to an eviction.

3.10.1 ObliviStore

ObliviStore [SS13] is a prominent example of a partition based ORAM scheme. It is based on the SSS ORAM by Stefanov et al. [SSS11], but with several modifications and innovations that allow it to support asynchronous operations. This is a huge benefit in cloud storage scenarios, since every request experiences some latency. This latency depends on many factors, including the physical distance to the cloud server and the current load on the server handling the request. Some requests can therefore take longer to complete than others. ORAM schemes that allow asynchronous operations can use this time to process other requests instead of sitting idle while waiting for requests to complete.

A drawback of allowing asynchronous operations is that it introduces a new channel that can leak information to an adversary; the *timing* channel. To ensure obliviousness, the timing of asynchronous requests must not depend on the timing of the clients original request to the ORAM. The authors of ObliviStore formalized this by defining the notion of *oblivious scheduling* and showed that ObliviStore satisfies this requirement. Their definition is restated below:

Definition 3.2. (Oblivious accesses and scheduling)
Let seq_0 and seq_1 denote two data access sequences of the same length and with the same timing:

$$seq_0 := [(blockid_1, t_1), (blockid_2, t_2), ..., (blockid_m, t_m)]$$

$$seq_1 := [(blockid'_1, t'_1), (blockid'_2, t'_2), ..., (blockid'_m, t'_m)]$$

Define the following game with an adversary who is in control of the network and the storage server:

– The client flips a random coin b.

– Now the client runs a distributed asynchronous ORAM algorithm and plays access sequence seq_b with the adversary.

– The adversary observes the resulting event sequence and outputs a guess b' of b.

We say that a (distributed) asynchronous ORAM is secure, if for any polynomial-time adversary, for any two sequences seq_0 and seq_1 of the same length and timing, $|\Pr[b' = b] - \frac{1}{2}| \leq negl(\lambda)$, where λ is a security parameter, and $negl$ is a negligible function. Note that the set of events observed by the adversary in the non-distributed and distributed case are given in Equations 1 and 2 respectively.

To meet this requirement, a lot of complexity was introduced to the design of ObliviStore (e.g 4 semaphores are required to properly synchronize all tasks). This makes implementation difficult. Bindschaedler et al. [BNP$^+$15] discovered a subtle issue in the authors implementation of ObliviStore that undermines its theoretical security claim.

3.10.2 CURIOUS

CURIOUS is a general partition based framework, proposed by Bindschaedler et al. [BNP$^+$15]. The authors observed that metrics like monetary expense and response time had not been considered in existing ORAM schemes. Because of this they created CURIOUS, which while being asymptotically worse than existing schemes like ObliviStore, performs better under these metrics.

Another benefit of CURIOUS is that it is modular. While ObliviStore uses a variant of the SSS ORAM [SSS11] for its partitions, CURIOUS allows using any

ORAM scheme. This means that its performance can be continually improved as new and faster ORAM schemes are discovered. It also reduces complexity, making implementation easier. In this way some of the issues plaguing ObliviStore can be avoided. The authors were able to prove that this simplified construction also satisfies the oblivious scheduling requirement.

3.11 Differentially Private ORAMs

Differentially Private ORAMs aim to improve the performance of ORAM by sacrificing some of the privacy guarantees provided by the original ORAM definition. This is done by adopting a new definition based on *differential privacy*. The notion of differential privacy was first introduced by Cynthia Dwork [Dwo06] as a way to quantify the privacy of users whose data has been collected for statistical purposes. Wagh et al. [WCM16] used this to propose a new ORAM definition where complete privacy is no longer a requirement. Instead, schemes can be designed to target different levels of privacy, opening up a new design space that had previously not been considered by the research community.

These schemes are not directly comparable to traditional ORAM schemes, since they do not adhere to the traditional definition of ORAM. Because of this, they are not considered further in this thesis. The interested reader is referred to the paper by Wagh et al. [WCM16] that originally proposed the idea for more information.

3.12 Oblivious Parallel RAMs

The previously mentioned ORAM schemes are not parallel from the point of view of the user of the ORAM. This means that the ORAM can not be accessed by more than one user at the same time. A branch of ORAM research has focused on addressing this issue and designing ORAM schemes that can support this, known as Oblivious Paralell RAMs (OPRAMs). These schemes are not further discussed in this thesis, but the interested reader is referred to [BCP16] for more details.

3.13 Summary

Table 3.1 provides a high-level overview of the ORAM schemes mentioned in this chapter. In addition, the theoretical bandwidth overhead and local storage requirements of each scheme is specified, using asymptotic notation.

Scheme	Type	Overhead	Local Storage
Trivial ORAM [GO96]	Trivial	$O(N)$	$O(1)$
Square-Root ORAM [GO96]	Square-Root	$O(\sqrt{N} \log N)$	$O(1)$
Hierarchical ORAM [GO96]	Hierarchical	$O((\log N)^3)$	$O(1)$
PracticalOS [GMOT12]	Large-Message	$O(1)$	$O(2\sqrt{N})$
PathORAM [SvDS+13]	Tree-Based	$O(\log N)$	$O(\log N) \cdot \omega(1)$
RingORAM [RFK+15]	Tree-Based	$O(\log N)$	$O(\log N) \cdot \omega(1)$
ObliviStore [SS13]	Partition-Based	$O(\log N)$	$O(N)$
CURIOUS [BNP+15]	Partition-Based	Tunable	Tunable

Table 3.1: Comparision of the ORAM schemes mentioned in this chapter.

Chapter 4 Methodology

The methodology used in this thesis is similar to the methodology used by Bindschaedler et al. [BNP$^+$15]. However, to avoid the problems discussed in Section 1.2, some modifications have been made. At a high level, the methodology consists of five steps:

1. Selecting representative ORAM schemes.
2. Implementing the test framework and schemes.
3. Generating workloads.
4. Performing tests with and without ORAM.
5. Evaluating the results and drawing conclusions.

Each step is explained in detail below. While all of these steps draw inspiration from [BNP$^+$15], the second and third step have been changed significantly. These steps are therefore explained in more detail.

4.1 Selecting schemes

Three criteria were used when selecting ORAM schemes. First of all, schemes should be applicable to current cloud storage services. This means that they do not rely on features not currently provided by mainstream cloud storage APIs. An example of this is server-side computations. Most cloud storage services are limited to uploading/downloading files and do not provide any mechanisms for doing computations in the cloud. Some ORAM schemes are designed around server-side computations (e.g [DvDF$^+$16]) while others use them as an optional optimization (e.g RingORAM [RFK$^+$15]) to increase performance in applications where they are available. The former are not considered in this thesis, while the latter are considered without any server-side optimizations.

The second criteria is that schemes should adhere to the mainstream definition of ORAM (see Section 3.1). This is to ensure that results are comparable. Examples of schemes that do not adhere to this are Differentially Private ORAMs (DP-ORAMs) and OPRAMs (see Sections 3.11 and 3.12).

Finally, schemes should have existing implementations available or a sufficiently detailed description so that they can be implemented in the timeframe of the thesis. This is a purely practical criterion, based on the experiences of Bindschaedler et al. [BNP$^+$15]. In their paper they noted that several schemes were difficult to implement because "the papers did not describe many important design details for developing working systems."

With the above criteria in mind, the following schemes were selected:

Name	Type	Proposed by
ObliviStore	Partition-Based	Stefanov & Shi [SS13]
CURIOUS	Partition-Based	Bindschaedler et al. [BNP$^+$15]
RingORAM	Tree-Based	Ren et al. [RFK$^+$15]

Table 4.1: The ORAM schemes that were selected for testing.

PracitcalOS (by Goodrich et al. [GMOT12]) was also considered, since an implementation was available. However, limited testing showed that this scheme incurred very high costs. The thesis had a limited budget ($200 on the IBM Cloud) and it was therefore not possible to include it.

4.2 Implementation

This section describes how the software components that were needed to run tests were implemented. First the implementation of the testing framework and the ORAM proxy is described, followed by the implementations of the ORAM schemes that were tested.

Test framework / ORAM proxy

The test framework was implemented in Java by extending the code for the CURIOUS framework[1] published by Bindschaedler et al. [BNP$^+$15]. This codebase contains all the code they used in to run their tests, as well as implementations of the ORAM schemes they tested. The code is modular, meaning that new features can be added without large modifications to the existing codebase. This was exploited to add two

[1]CURIOUS framework, http://seclab.soic.indiana.edu/curious/

new major features to the test framework. These additions are described in detail below.

The first feature that was added was support for running tests on OpenStack Swift. This was implemented using JOSS[2], a Java library that provides a Swift API client. The reason for adding this feature was that the the original test setup of the thesis used a private Swift cloud. This cloud was later replaced by the IBM Cloud because of performance issues.

The second feature was added to address problems with the methodology of Bindschaedler et al. [BNP+15]. Their tests are based on first running an application, in this case Filebench[3], on a bucket in Amazon S3. When the execution of the application finishes, a log of the requests to the bucket is collected. The requests in the log are then replayed through an ORAM scheme. This approach allows the ORAM scheme to be separated from the application and tests of the ORAM schemes to be run independently of the application itself. While this might seem attractive, this solution can lead to inaccuracies when using ORAM schemes that can handle parallel requests (see Section 1.2).

This thesis will therefore use a different approach; running the ORAM schemes in a proxy. This can be implemented by mimicking the API of an existing cloud service and translating each request into a series of ORAM requests. This solution is flexible, since only minimal modification to the application is required. It also avoids the inaccuracies related to parallel ORAM schemes, since the application runs at the same time as the proxy. This approach is illustrated in Figure 4.1.

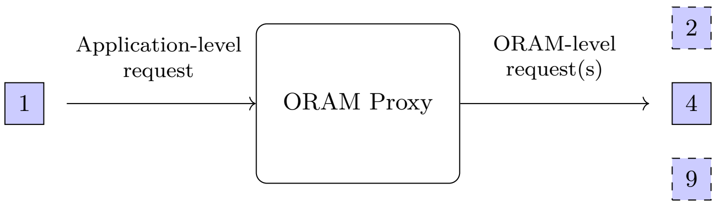
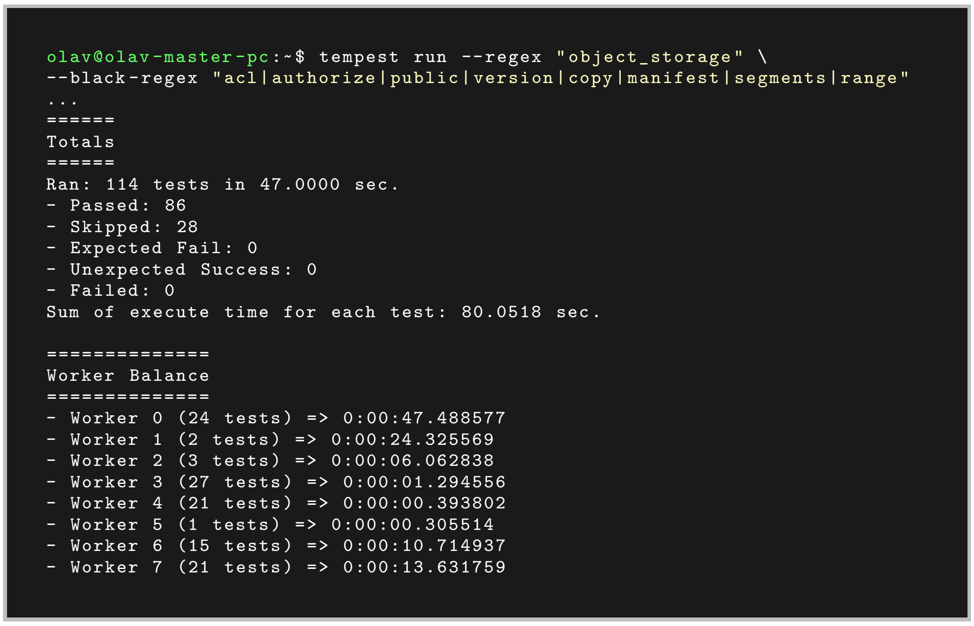


Figure 4.1: An ORAM proxy translating an application level request into one or more ORAM requests.

This "proxy approach" was the second feature added to the codebase. It was implemented by creating an implementation of the Swift API. This implementation was created from scratch since existing implementations (e.g JOSS) only supported the client side of the API. The public documentation[4] of the API was used during

[2]JOSS, http://joss.javaswift.org/
[3]Filebench, https://github.com/filebench/filebench
[4]OpenStack Swift API, https://developer.openstack.org/api-ref/object-store/

development. To ensure the correctness of the implementation, a tool called *Tempest*[5] was used. This tool provides a set of integration tests that can be used to test deployments of OpenStack. Figure 4.2 shows the output when running tempest against the ORAM proxy. The flag `--regex` is used to specify which tests should be included, in this case only tests matching the string "object_storage". By providing this flag, only tests for Swift will run. The flag `--black-regex` does the exact opposite. It specifies which tests tempest should not run. In this case it is used to exclude tests for the more advanced features of the API. These features were not fully implemented, since they are superfluous when running ORAM tests.



```
olav@olav-master-pc:~$ tempest run --regex "object_storage" \
--black-regex "acl|authorize|public|version|copy|manifest|segments|range"
...
======
Totals
======
Ran: 114 tests in 47.0000 sec.
- Passed: 86
- Skipped: 28
- Expected Fail: 0
- Unexpected Success: 0
- Failed: 0
Sum of execute time for each test: 80.0518 sec.

==============
Worker Balance
==============
- Worker 0 (24 tests) => 0:00:47.488577
- Worker 1 (2 tests) => 0:00:24.325569
- Worker 2 (3 tests) => 0:00:06.062838
- Worker 3 (27 tests) => 0:00:01.294556
- Worker 4 (21 tests) => 0:00:00.393802
- Worker 5 (1 tests) => 0:00:00.305514
- Worker 6 (15 tests) => 0:00:10.714937
- Worker 7 (21 tests) => 0:00:13.631759
```

Figure 4.2: Verifying the correctness of the Swift API implementation using tempest.

Another improvement that was made to the codebase is adding support for requests of arbitrary sizes. This was not considered in the study by Bindschaedler et al. [BNP+15], which used workloads where the file sizes were always aligned with the block size they had chosen. This is not the case for the workloads used in this thesis or for real-world cloud applications. To be able to support this, a padding scheme was added. Application-level requests are broken up into blocks. For each block, at least one byte is reserved for a special value that marks the end of the data (`0x01` was chosen for this value). If there are any unused bytes after this value, they are set to (0x00). When reading blocks, the program starts from the last byte and reads

[5]Tempest, https://github.com/openstack/tempest

backwards. Once it reaches the special value, it knows that the padding has ended and the remaining data is the application-level data.

Schemes

Implementations for the selected ORAM schemes (Table 4.1) were adapted from various sources. The implementations for ObliviStore, CURIOUS and PracticalOS were adapted from the code released by Bindschaedler et al. [BNP+15]. Only minor changes had to be made to these schemes. Note that the implementation of PracticalOS was only used for a limited time during development since it was dropped early on. The implementation of RingORAM was adapted from a publicly available repository[6] published by the GitHub user `caohuikang`. A number of changes had to be made both to this implementation and to the test framwork / ORAM proxy to make it work. One of these changes was adding support for partial downloads, since RingORAM needs to be able to fetch only a single block from a bucket (without this ability the bandwidth improvements of RingORAM are lost). This is supported by most cloud APIs, but was not originally suppported in the code by Bindschaedler et al. [BNP+15], since none of the schemes they tested required this ability.

Challenges

This section describes some of the challenges faced during the implementation phase. In addition to adding new features, some bugs in the testing framework had to be fixed. An example of this is the way that asynchronous requests were handled. In the original code a `HashMap`[7] was used to store pending requests. `HashMap` is a data structure in the Java standard library designed for storing key-value pairs. The problem with using this class is that it does not support concurrency, meaning that multiple requests arriving at the same time will crash the program. This bug was likely never encountered in the original study, since requests were always sequentially read from a log file. However, with the "proxy approach" requests can arrive at any time from the software using the ORAM, triggering the bug. To fix the bug, it was sufficient to replace the `HashMap` with a `ConcurrentHashMap`, another class from the Java standard library that provides the same functionality while also allowing concurrent accesses. This fix is shown in Figure 4.3.

Another challenge that had to be tackled was high memory usage. The ORAM proxy needed to be able to handle multiple application-level requests arriving at the same time. To support this, asynchronous Java Servlets[8] were used in conjunction with a queue of pending application-level requests. When running the first tests, it was discovered that this queue would not be cleared as the application-level requests

[6]RingORAM, https://github.com/caohuikang/RingORAM
[7]HashMap, https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
[8]Java Servlets, https://www.oracle.com/technetwork/java/index-jsp-135475.html

were completed. This lead to a *memory leak*, where some tests would consume all the memory available on the test computer (32 GB). This issue and a number of other smaller issues related to concurrency had to be fixed before tests could be run.

```
--- a/src/main/java/eoram/cloudexp/evaluation/PerformanceEvaluationLogger.
    java
+++ b/src/main/java/eoram/cloudexp/evaluation/PerformanceEvaluationLogger.
    java
- protected Map<Long, RequestLogItem> requestsMap = new HashMap<Long,
    RequestLogItem>();
+ protected Map<Long, RequestLogItem> requestsMap = new ConcurrentHashMap<
    Long, RequestLogItem>();
```

Figure 4.3: Example of a change made to the codebase to fix issues with asynchronicity.

4.3 Workloads

As mentioned in Chapter 1, Bindschaedler et al. [BNP+15] conducted a study with goals similar to those of this thesis. One of the justifications for conducting a new study is that the workloads they used were not representative of real cloud applications (see Section 1.2). For this reason, the workloads used in this thesis were crafted to closely resemble real-world use cases.

4.3.1 Synthetic Workloads

To this end, two classes of workloads were selected. One representing a *home user* and one representing an *organization*. For each class, a small number of workloads were created. It is hoped that these workloads cover the most common cloud storage use cases where the security benefits of ORAM could be of interest. The workloads are summarized in Table 4.2.

Workload	Characteristics	Distribution	File Sizes
A: Home user	read often, write often	R: 40%, W: 60%	1 KB - 4 GB
B: Traveling user	read often, write once	R: 99%, W: 1%	1 KB - 4 MB
C: Email backup	read rarely, write often	R: 1%, W: 99%	10 MB - 25 MB
D: Code distribution	read often, write rarely	R: 80%, W: 20%	1 MB - 320 MB
E: E-Government	even reads, bursty writes	variable	1 KB - 11 MB

Table 4.2: Summary of the workloads that were tested.

Home User

In the home user case, two workloads were constructed. The first workload represents a home user that stores all of their personal files in the cloud. This means that files will vary greatly in size, from small text documents to large video files. Liu et al. [LHFY13] studied the usage patterns of personal cloud storage, by recording a five-month access trace of a campus cloud storage system with approximately 19,000 users. They found that file sizes followed a bimodal distribution with peaks around 1 KB and 4 MB. They also found that 90% of files were smaller than 4 MB and a neglible amount of files were larger than 4 GB. In addition, they found that the ratio of read to write operations was 0.65. Based on this information, the workload was set to use files between 1 KB and 4GB, chosen randomly according to Figure 3 in their paper. The access pattern was set to 40% reads and 60% writes. This workload is referred to as *Workload A*, or the *home user* workload.

The second workload represents a home user that uses the cloud to store files they need while traveling. In this case the files are uploaded once and new files are very rarely added. The files still vary in size, but they are assumed to be smaller since they need to be accessible over constrained links such as mobile networks. To represent this, the first peak of the distribution found by Liu et al. [LHFY13] is used. File sizes were therefore set to range from 1KB to 4MB. The files were chosen according to the same distribution as Workload A, but with a cutoff at 4 MB. This workload is referred to as *Workload B*, or the *traveling user* workload.

Organization

For the organization case, three workloads were created. The first workload represents an organization that uses cloud storage to backup large email attachments. In this case files can be assumed to be larger than a predefined size, chosen to be 10MB. The maximum size was chosen to be 25MB, based on the limit imposed by Gmail[9] at the time of writing. New files are regularly added, and old files are deleted. Files only need to be downloaded in exceptional cases (e.g after a system crash). This workload is referred to as *Workload C*, or the *email backup* workload.

The second workload represents an organization that shares code with other organizations using a version control system. The organization wants to take advantage of cloud storage to store their code, but does not want to reveal any information about the code to the provider of the cloud service. A realistic example of this could be a company that sells a Software Development Kit (SDK). The company uses a version control system in the development process and, to ease distribution of updates, customers have read access to this system. It is assumed that many

[9]Gmail attachment limit, https://support.google.com/mail/answer/6584

customers are using the SDK and that the version control system therefore sees more reads than writes.

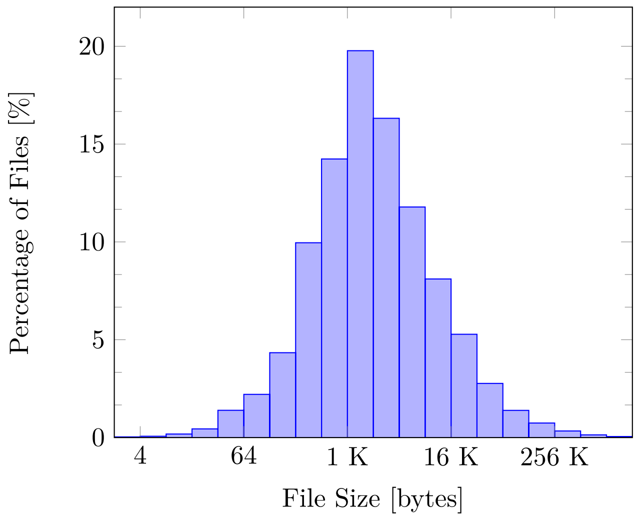


Figure 4.4: File size distribution of popular GitHub repositories.

To model this scenario, a study of software projects using the popular version control system Git[10] was conducted. For this study, a selection of 100 repositories was downloaded from GitHub[11], an online service that provides free hosting of Git repositories for open-source projects. This process was automated in a script using GitHub's API[12]. The repositories were selected based on popularity, with 10 repositories being downloaded for each of the 10 most popular programming languages on the site.

Once the repositories were downloaded, a separate script was used to collect information about file and folder sizes for each repository. This data was used to form two datasets. The first dataset, referred to as the *file size dataset*, contains the size of every file in all the downloaded repositories. This dataset is visualized as a histogram in Figure 4.4. The second dataset, referred to as the *repository size dataset*, contains the total size of each repository (i.e the sum of the file sizes, grouped by repository). Key statistics about the two datasets is provided in Table 4.3.

These datasets were used to select the parameters for the workload. File sizes were set to range from 0.01 KB to 10 MB (based on the median and standard deviation

[10]Git, https://git-scm.com/
[11]GitHub, https://github.com
[12]Github API, https://developer.github.com/v3/

of the file size dataset) and chosen according to the distribution shown in Figure 4.4. This workload is referred to as *Workload D*, or the *code distribution* workload.

	File Size [KB]	Repository Size [MB]
Mean	49.67	187.14
Median	1.84	36.84
Standard deviation	5151.06	430.58
Sample minimum	0	0.37
Sample maximum	2581568.38	3534.51

Table 4.3: Key statistics about the GitHub repositories that were collected.

The third workload was designed to mirror a cloud service that has to handle very bursty traffic. Such services are characterized by long periods of low to medium traffic, followed by intermittent periods of very high traffic. Examples of such services are E-Government services, where the high traffic periods coincide with events like elections or tax returns. A good example of this is the Norwegian E-Government system Altinn, which handles tax returns. The system had to be taken offline both in 2011 and 2012 when the yearly tax statements were published due to excessive network traffic [Hol16].

The workload was created to represent a hypothetical E-Government service that handles applications from citizens. The applications can contain attachments, usually digital or scanned documents, which are stored in the cloud. It is assumed that applications have a fixed deadline, meaining that the service will experience a spike in traffic around the deadline. To model this service, the workload was set to have multiple stages. Each stage has a different distribution of read and write operations, with the stage representing the spike in traffic having a much higher proportion of writes. File sizes were set to range from 1KB to 11MB, based on a study by Hawa et al. [HRAAN12] that found that document files shared on the BitTorrent network had a median size of 0.11 MB, with a standard deviation of 10.7 MB. This workload is referred to as *Workload E*, or the *E-Government* workload.

4.3.2 Real Workloads

In addition to the synthetic workloads, a small number of real-world tests were also conducted. These tests are based on Workload D, both for practical reasons (the availability of existing cloud applications that can be used with the ORAM proxy) and because of the results of the synthetic tests (for certain schemes and block sizes, low response times and slowdowns were achieved).

Since Workload D is based on a scenario where version control is used to distribute code, *Git* was used to perform the tests. For each of the ORAM schemes that were tested, four steps were carried out and the time it took to complete each step was recorded. The steps are listed below (see Section 2.3 for an explanation of the Git terminology):

1. Pushing a repository to the server for the first time.
2. Cloning the repository for the first time.
3. Pushing a new commit to the server.
4. Pulling a new commit after having already cloned the repository.

The open-source project *Dulwich*[13] was used for the Git server. Dulwich is a Python implementation of the Git protocol and comes with the necessary scripts to set up a Git server. The server supports using Swift as a storage backend, making it straightforward to use it with the ORAM proxy.

The OpenStack SDK's Git repository[14] was used as a representative repository. At the time of writing the repository had 5894 commits from 238 contributors and totaled 6.3 MBs in size when cloned. This repository was chosen because it represents the scenario considered in Workload D (an SDK that is distributed using version control) and is publicly available.

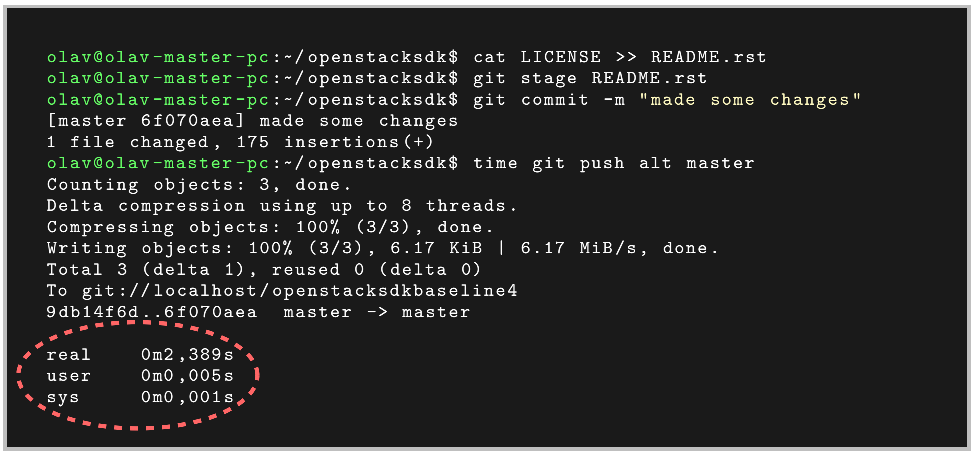
These tests were carried out for CURIOUS with a block size of 16 KB and ObliviStore with a block size of 64 KB. These schemes were selected based on the results from Workload D. Additional schemes and block sizes were not tested due to time and resource (IBM cloud credit) restrictions.

To measure the time used by each step, the `time`[15] command-line utility was used. This utility takes a command as input, runs the command and waits for it to finish. When the command finishes the time it took to execute is printed (both real time and CPU-time). An example of this is shown in Figure 4.5.

4.4 Test setup

Each of the workloads presented in Section 4.3 were tested in the public-cloud scenario described in Section 1.1. In addition, the Git workloads were tested in the private-cloud scenario. Figures 4.7 and 4.8 show the test setup in each case.

[13]Dulwich, https://github.com/dulwich/dulwich
[14]OpenStack SDK, https://github.com/openstack/openstacksdk
[15]time, https://linux.die.net/man/1/time



```
olav@olav-master-pc:~/openstacksdk$ cat LICENSE >> README.rst
olav@olav-master-pc:~/openstacksdk$ git stage README.rst
olav@olav-master-pc:~/openstacksdk$ git commit -m "made some changes"
[master 6f070aea] made some changes
1 file changed, 175 insertions(+)
olav@olav-master-pc:~/openstacksdk$ time git push alt master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 6.17 KiB | 6.17 MiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To git://localhost/openstacksdkbaseline4
9db14f6d..6f070aea  master -> master

real    0m2,389s
user    0m0,005s
sys     0m0,001s
```

Figure 4.5: Timing a Git operation using the `time` command-line utility.

Client

All tests were run on a desktop computer provided by NTNU. The specifications of the computer is shown in Table 4.4.

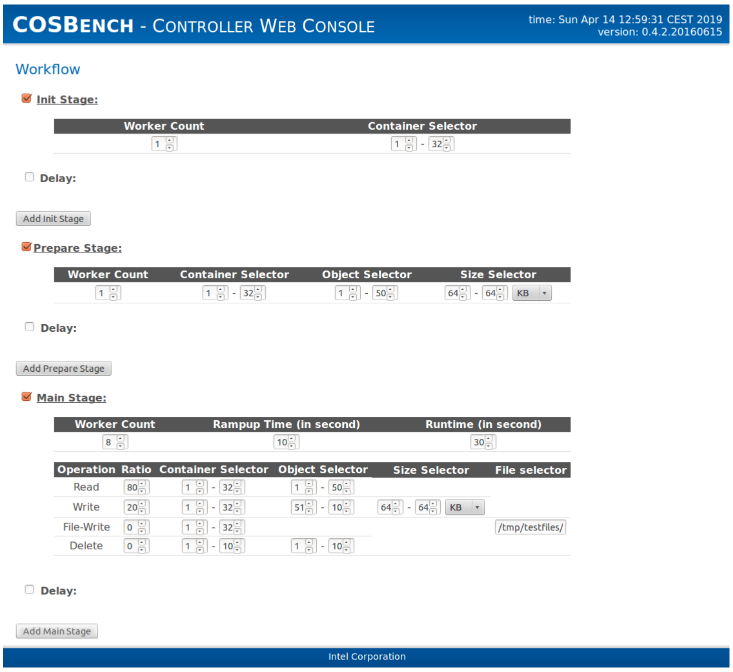
Operating System	Ubuntu 18.04 LTS
CPU	Intel® Core™ i7-6700 CPU @ 3.40GHz
RAM	32GB (4 × 16ATF1G64AZ-2G1B1 2133MHZ)
Storage	512GB (SK Hynix PC300 NVMe SSD)
Network	Intel® Ethernet Connection I219-LM

Table 4.4: Specifications of the desktop computer used in all tests.

To generate the synthetic workloads *COSBench* [ZCW+12], a cloud storage benchmarking tool created by researchers at Intel, was used. This tool is designed with the strengths and limitations of cloud storage in mind and can therefore provide more realistic workloads compared to traditional tools like Filebench. It can also be used with cloud storage APIs directly, avoiding the extra steps of mounting the cloud storage as a folder that were taken by Bindschaedler et al. [BNP+15] The concerns raised in Section 1.2 are therefore addressed.

While COSBench is a step in the right direction, the workloads are still synthetic. This means that there will be a gap between the results obtained using COSBench and results obtained using real cloud applications. This is unfortunate, but necessary because of time constraints on the thesis. Future works could improve this by finding or developing real-world cloud applications and use them instead of COSBench.

Workloads in COSBench are specified in Extensible Markup Language (XML) files. These files can either be created manually or generated from the COSBench web interface. Workloads are divided into one or more *work stages*. A work stage typically specifies the types of operations that will be performed (e.g read, write) and the distribution of those operations (e.g 90% read, 10% write). There are four special work stages (`init`, `prepare`, `cleanup`, and `dispose`) that are responsible for the operations that need to be performed before and after a test. The full workload format is described in the COSBench user guide [Wan14].


COSBENCH - CONTROLLER WEB CONSOLE

time: Sun Apr 14 12:59:31 CEST 2019
version: 0.4.2.20160615

Workflow

✓ Init Stage:

Worker Count	Container Selector
1	1 - 32

☐ Delay:

Add Init Stage

✓ Prepare Stage:

Worker Count	Container Selector	Object Selector	Size Selector
1	1 - 32	1 - 50	64 - 64 KB

☐ Delay:

Add Prepare Stage

✓ Main Stage:

Worker Count	Rampup Time (in second)	Runtime (in second)
8	10	30

Operation	Ratio	Container Selector	Object Selector	Size Selector	File selector
Read	80	1 - 32	1 - 50		
Write	20	1 - 32	51 - 10	64 - 64 KB	
File-Write	0	1 - 32			/tmp/testfiles/
Delete	0	1 - 10	1 - 10		

☐ Delay:

Add Main Stage

Intel Corporation

Figure 4.6: Creating a workload using the COSBench web interface.

The workloads used in this thesis were first created in the web interface. The XML-files were then manually tweaked if necessary. XML-files for all workloads are included in Appendix A.

Cloud

IBM Cloud Object Storage[16] was used as the cloud storage service in all tests. This solution was chosen in favor of the alternatives for economical reasons. IBM provides

[16]IBM Cloud Object Storage, https://www.ibm.com/cloud/object-storage

$200 free credits for new users, which made it possible to run more tests than what would otherwise be possible with the limited budget of the thesis. The IBM Cloud Object Storage has a number of regions available, allowing data to be stored in different physical locations. The `eu-gb-standard` region was used for all tests.

The quality of the link between the desktop computer and the cloud, was measued using `iperf`[17], a command line tool for active measurements of bandwidth, loss and other network parameters. The average bandwidth of the link was measured to 510.4 Mbits/sec, with a jitter of 0.016 ms and a packet loss of 0.02%. The average Round Trip Time (RTT) was measured to 23.6 ms using the `ping`[18] command line tool.

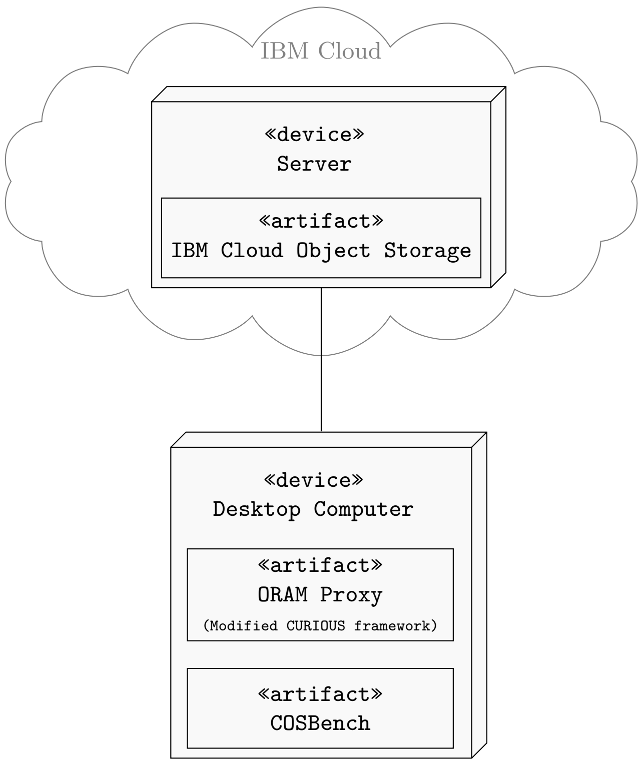


Figure 4.7: Deployment diagram showing the test setup used for the public cloud scenario.

The IBM Cloud was also used to represent the trusted cloud server in the private cloud scenario. A virtual machine was provisioned for this purpouse. The virtual machine was allocated ample resources (4 virtual CPUs and 32 GBs of RAM) so that it would not be the bottleneck in the system.

[17]iperf, https://iperf.fr/
[18]ping (a part of iputils), https://github.com/iputils/iputils

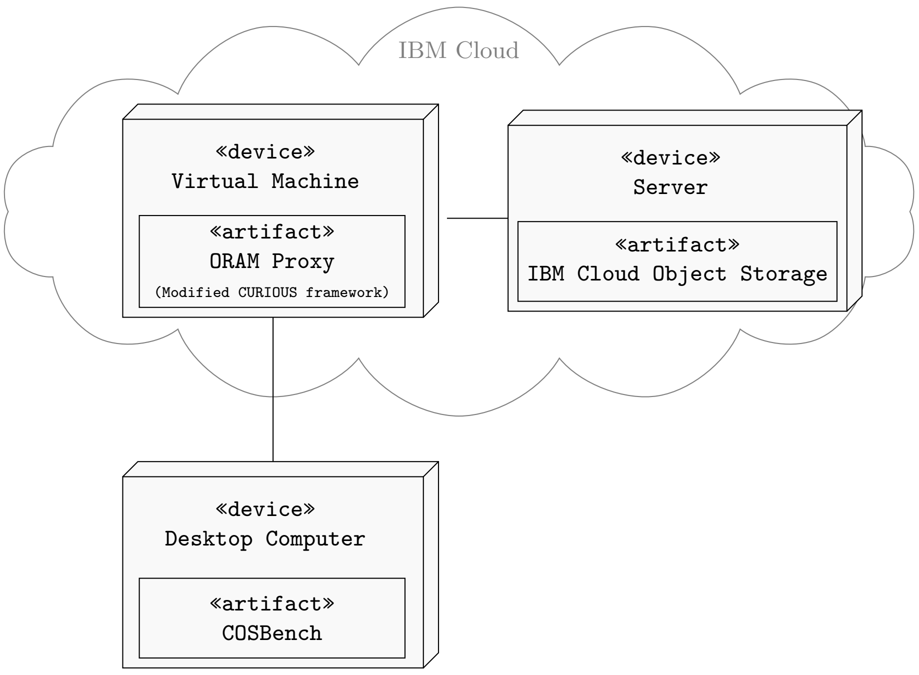


Figure 4.8: Deployment diagram showing the test setup used for the private cloud scenario.

Moving the ORAM functions from the client to the virtual machine in the cloud was an easy task, thanks to the "proxy approach" adopted during the implementation phase. Since the ORAM schemes are implemented in a proxy, they are independent of both the application and the cloud storage. The ORAM proxy cloud therefore be installed on the cloud server and the application (e.g Git for the real workloads) could be configured to connect to its Internet Protocol (IP) address. No changes to the code were required to support the private cloud scenario.

Block sizes

One of the limitations of the Bindschaedler et al. [BNP+15] study is the limited number of block sizes they tested, with most workloads only being tested for two block sizes and all block sizes being in the KB range. To improve upon this, a wider range of block sizes were tested in this thesis. For each workload three block sizes were selected. This was the widest range possible within the given budget. The three block sizes were chosen based on some preliminary tests, where block sizes ranging from 4 KB to 16MB were tested for each workload. These tests were carried out without using ORAM. The results of these preliminary tests were used to select the most economical block sizes to use for the full tests.

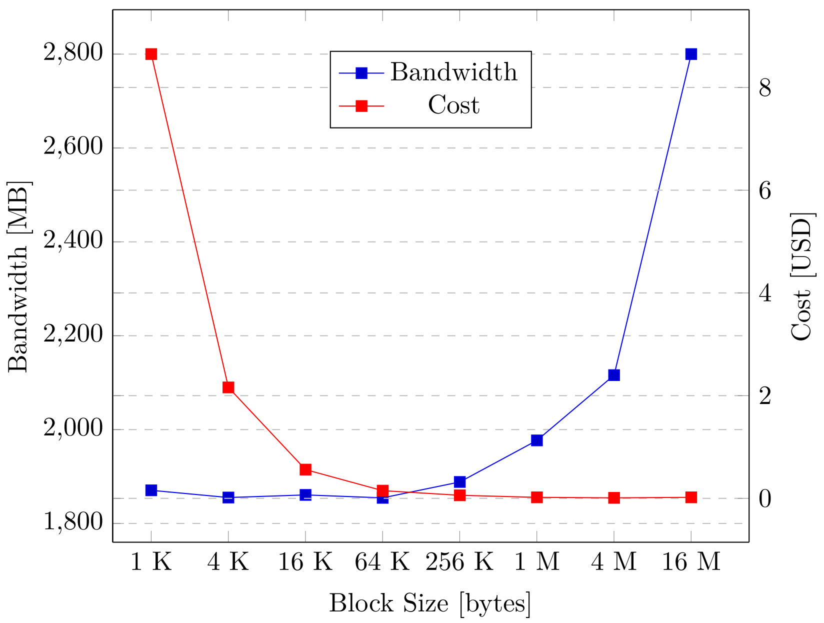


Figure 4.9: Plot showing the bandwidth requirements (in MB) and the estimated cost (in USD) of running Workload C on the IBM Cloud for different block sizes. To better capture the effect of block size in isolation, the estimation was made without ORAM.

Figure 4.9 illustrates the results that were used to select block sizes for Workload C. In this figure the relationship between bandwidth and cost is highlighted. An interesting observation here is that, as the block size increases, the cost goes down, but the bandwidth goes up. This was true for all of the workloads that were tested, and is a consequence of the pricing model used by the IBM Cloud (a high number of requests is more expensive than high bandwidth).

Parameters

Two of the ORAM schemes selected for the tests, namely CURIOUS and RingORAM, are tunable. This means that one or more parameters can be tuned to different application scenarios. In this thesis, the parameters were set for a high local-storage scenario. For CURIOUIS this meant setting $b = 3$ and $z = 5$. These are the same parameters that Bindschaedler et al. [BNP$^+$15] used when they evaluated the scheme and are taken from Table 7 in their paper. For RingORAM the parameters Z (real block count) and A (eviction rate) were set to 33 and 48 according to Table 5 in the RingORAM paper [RFK$^+$15], which assumes a large client storage budget. The S (dummy block count) parameter was calculated using the analytical model provided in Table 4 in the same paper. This resulted in a value of $S = 63$.

4.5 Evaluation

To evaluate the results, the metrics identified by Bindschaedler et al. [BNP+15] were used. These metrics are:

Bandwidth. This is the metric that has traditionally been used to evaluate ORAM schemes. This metric captures the total number of bytes transfered between the client and the storage for a given workload. Bindschaedler et al. [BNP+15] argued that this metric is less important for practical applications, however in certain pricing models (such as the one used by Rackspace Cloud Files, see Table 4.5) it can have a big effect on the cost and thus on the practicality of using ORAM.

Response time. The time it takes from an application-level request is issued until the response reaches the application is captured by the response time. This metric can have a large impact on the performance of real-world applications, depending on the ability of the application to process requests concurrently. Applications that issue requests synchronously need a low response time to be pracitcal, while applications that can process requests asynchronously can tolerate larger response times. An ORAM schemes ability to process requests asynchronously also affects practical performance. Synchronous schemes, like RingORAM, often exhibit low response times when the block size closely matches the average size of the application-level requests. However, when smaller block sizes are used the response times increase dramatically. This is because more than one ORAM-level request has to be completed before the application-level request is completed. If an ORAM scheme processes requests sequentially, the response time of an application-level request becomes the sum of the response times of the required ORAM-level requests.

Slowdown. The ratio of time taken to complete a workload using a particular ORAM scheme, compared to completing the same workload without using ORAM, is measured by the slowdown. This metric captures the effects of asynchronicity better than the response time, since it includes the potential benefits of processing multiple application-level requests in parallel. This is not captured by the response time, as it only considers a single application-level request.

Outsource ratio. The amount of local storage required by an ORAM scheme affects both the practicality and the performance of the scheme. Some schemes use a predetermined amount of local storage, often dependent on the size of the remote storage, while other schemes (e.g RingORAM) support tuning the amount of local storage for the intended application scenario. Bindschaedler et al. [BNP+15] studied the effects of minimizing local storage in a cloud setting and found that the cost of doing this, in terms of slowdown and response time was high. Because of this and other practical factors (e.g modern smartphones having several GBs of RAM) they argued that the absolute number of bytes is a bad metric for local storage in cloud scenarios. They proposed the *outsource ratio*, i.e the ratio of local storage to remote storage, as a better metric.

Monetary expense. Cloud storage services typically operate on a pay-as-you-go basis. This means that the additional resources (e.g bandwidth, upload/download operations, etc.) required when using ORAM have a direct impact on the bill the user pays. One of the main drivers for cloud storage adoption is cost-savings [Avr14]. Because of this, it is cruical for practical applications that the additional costs of ORAM does not outweigh the benefits of using cloud storage.

Elasticity and reliability. An important feature of cloud services are their ability to scale. The use of ORAM can have an effect on this ability. It is therefore important that the scalabilty meets the need of the application, even when ORAM is used. The elasticity metric captures this ability. Another key feature of cloud storage services is strong reliability guarantees, with data being replicated across multiple servers. ORAM schemes usually keep some important state (e.g position map, keys for encryption, etc.) locally. This data can be lost if the client crashes, affecting the reliability of the data. It is therefore important for practical applications that the local state is properly backed up. This ability is captured by the reliability metric. Both of these metrics are more qualitative in nature and can not be directly measured from tests.

Some of these metrics (e.g *bandwidth*, *response time*) were available directly from the COSBench results, while others (e.g *monetary expense*, *outsource ratio*) were obtained indirectly. To calculate the monetary expenses, prices for a selection of popular cloud storage services were collected. The results are shown in Table 4.5. These prices were up to date as of June 2019. The qualitative metrics (*elasticity* and *reliability*) were not considered in this thesis. A new metric, not considered by Bindschaedler et al., was also measured and used in the evaluation:

Computational cost. Using cloud storage is normally computationally inexpensive. However, when ORAM is used additional, computations are required. The requirements differ from scheme to scheme, but a common overhead is the encryption and decryption of blocks. Shuffling blocks is another potentially expensive operation. Depending on the implementation, a shuffle can require copying large amounts of data around in the local storage. In some practical applications, it is cruical that the number of local computations is kept at a minimum. An example of this is smartphone applications, where computations direcly affect the battery life and thus the practicality of the application. The *computational cost* metric captures this.

	Amazon S3[19]	Google Cloud Storage[20]	Rackspace Cloud Files	IBM Object Storage[21]	Oracle Object Storage
Price per GB[22]	\$0.023/month	\$0.020/month	\$0.10/month	\$0.022/month	\$0.0255/GB
Bandwidth in	Free	Free	Free	Free	Free
Bandwidth out[23]	\$0.09/GB	\$0.11/GB[24]	\$0.12/GB	\$0.09/GB	Free[25]
GET requests	\$0.0004/1,000 reqs	\$0.0004/1,000 reqs	Free	\$0.0004/1,000 reqs	\$0.00034/1,000 reqs
PUT requests	\$0.005/1,000 reqs	\$0.005/1,000 reqs	Free	\$0.005/1,000 reqs	\$0.00034/1,000 reqs

Table 4.5: Prices for a selection of cloud object storage services. The data was gathered from the service providers' websites in April 2019 and were confirmed to be up to date as of June 2019.

[19] All prices are for the US East region
[20] All prices are for the `us-east1` region
[21] All prices are for the US East region, with Regional Standard storage
[22] Assuming less than 10TB of storage is used per month
[23] Assuming less than 10TB of bandwidth is used per month
[24] Assuming no egress bandwidth to Asia & Australia
[25] The first 10 TBs per month are free. A charge of \$0.0085 applies after this limit is exceeded.

Chapter 5 Results

This chapter presents the results of the experiments that were performed. First, the data gathered from running each of the synthetic workloads (A-E) is presented. The data from the real-world workloads (version control using git) is then presented. Note that this chapter only highlights a subset of the data that was gathered. The remaining data is available in Appendix C.

5.1 Synthetic workloads

In this section, the results from the synthetic tests are presented. For these tests, the metrics selected in Section 4.5 are used. Five of these metrics (bandwidth, response time, runtime, monetary expense and computational cost) were measured both for a baseline without ORAM and for each of the schemes selected in Section 4.1. Because of this, it was possible to calculate the relative overhead for each of these metrics. The overheads are dimensionless and represent the ratio between the baseline and the scheme in question (e.g a bandwidth overhead of 10 means that 10× more bandwidth was used, compared to the baseline). This section focuses on these overheads, with the raw values only being presented in interesting cases. The interested reader may refer to Appendix C, which contains both the raw and calculated values, while reading this section.

For each workload, the baseline results are presented first. Then the relative overheads for the three selected schemes are shown for each of the block sizes the workload was tested with. The remaining metric from Section 4.5, the outsource ratio, is not presented in this chapter, but is included in Appendix C.

Comparable (e.g two workloads that were tested using the same block size) and interesting results are highlighted whenever they are encountered. These results are further elaborated on and discussed in Chapter 6.

5.1.1 Workload A (Home User)

This section presents the results for Workload A (the home user workload). For this workload, tests were performed for block sizes of 64 KB, 256 KB and 1 MB.

Table 5.1 shows the baseline for this workload. The table is divided into two sections. The first section contains data that was collected during the test, while the second section contains relative metrics calculated from the data. The calculated metrics are all 1 in this case (since they are relative to the baseline), but are included for the sake of completeness.

Baseline (no ORAM, no blocks)	Download operations	29
	Upload operations	77
	MBs downloaded	29.02
	MBs uploaded	171.67
	Runtime (seconds)	10
	Average CPU usage (%)	4.97
	Peak CPU usage (%)	12.5
	Average RAM usage (MB)	112.72
	Peak RAM usage (MB)	145.58
	Read bandwidth (KB/s)	13849.52
	Write bandwidth (KB/s)	34352.56
	Avg. response time read (ms)	201.27
	Avg. response time write (ms)	296.33
	Cost (USD)	$0.01
	Bandwidth overhead	1
	Relative response time	1
	Relative slowdown	1
	Cost multiplier	1
	Outsource ratio	1

Table 5.1: Baseline results for Workload A. These results were measured by running the workload without using ORAM and without splitting the files into blocks.

The remaining results are summarized in Figure 5.1. The figure shows three bar charts, one for each block size. The charts are divided into four groups, one for each of the three ORAM schemes and one for running the workload without ORAM (but with files being split into blocks). Each group has five bars, one for each of the five overheads that were calculated. These are (from left to right): bandwidth overhead, cost overhead, relative response time, relative slowdown and computational overhead.

From these charts we can see that the overhead of using blocks (without ORAM) was relatively small in most cases. The bandwidth and cost overhead never exceeded 2× the baseline and were as low as 1.1× and 1× when 256 KB blocks were used. The relative response time and slowdown were higher, with the response time maxing out at 9.4× and the slowdown at 25×, both when 64 KB blocks were used. An interesting observation regarding these metrics is that when 64 KB blocks were used, they were higher for no ORAM than for ObliviStore or CURIOUS. Another interesting result is that the computational overheads were lower than the baseline, ranging from 0.7× to 0.9× the baseline. A possible explanation for this is that these tests were run synchronously (see Section 6.2), while the baseline was asynchronous. This could cause lower CPU usage as more time is spent waiting for replies from the server.

ObliviStore incurred high overheads for bandwidth and cost and relatively low overheads for the three other metrics. The bandwidth overhead ranged from 28.8× to 39.7×, with the overhead being highest when 256 KB blocks were used. ObliviStore had both the highest and the lowest cost overhead of all the ORAM schemes, with costs being as high as 192× the baseline when using 64 KB blocks and 36× when using 1 MB blocks. Interestingly, the cost overhead was at its peak when the bandwidth overhead was at its smallest. The relative response time and slowdown were more moderate, ranging from 3.5× to 3.9× and 8.5× to 10× the baseline value respectively. The computational overhead of using ObliviStore for this workload ranged from 6.5× to 7.2× the baseline value. The highest computational overhead was incurred when 256 KB blocks were used.

The overheads for CURIOUS are comparable to those of ObliviStore, but slightly higher in most cases. CURIOUS had the highest bandwidth overhead of all the schemes with the peak bandwidth being 94.8× higher than the baseline bandwidth. The cost of using CURIOUS ranged from 55× to 119× higher than the baseline value. For 256 KB and 1 MB blocks, this was higher than the cost of using ObliviStore, however for 64 KB blocks, the cost of using CURIOUS was significantly lower (119× vs 192× higher than the baseline). The relative response time and slowdown ranged from 3.5× to 3.9× and 8.5× to 10× respectively. These metrics were lowest when 256 KB blocks were used and highest when 64 KB blocks were used. The computational cost of CURIOUS was slightly higher than that of ObliviStore, ranging from 7.2× to 7.6× the baseline value.

Using RingORAM for this workload incurred high overheads for all metrics, except computational overhead. The bandwidth overhead ranged from 48.9× when using 1 MB blocks, to 82.7× when using 256 KB blocks. These overheads were smaller than those of CURIOUS for the same block sizes. The cost overhead of RingORAM was very close to that of CURIOUS, being only slightly higher for each of the three block sizes. The highest cost overhead was 120× higher than the baseline (compared to 119× for CURIOUS) and the lowest was 56× higher (compared to 55× for CURIOUS). The relative response time and slowdown of RingORAM were the highest of all the ORAM schemes, with the peak slowdown being 1440.3× the baseline. The computational overhead of RingORAM was low compared to the other schemes. It ranged from 0.8× for the smallest block size to 1.9× for the largest block size. An interesting observation regarding this is that the computational overhead of RingORAM when using 64 KB blocks was lower than both baselines (with and without blocks).

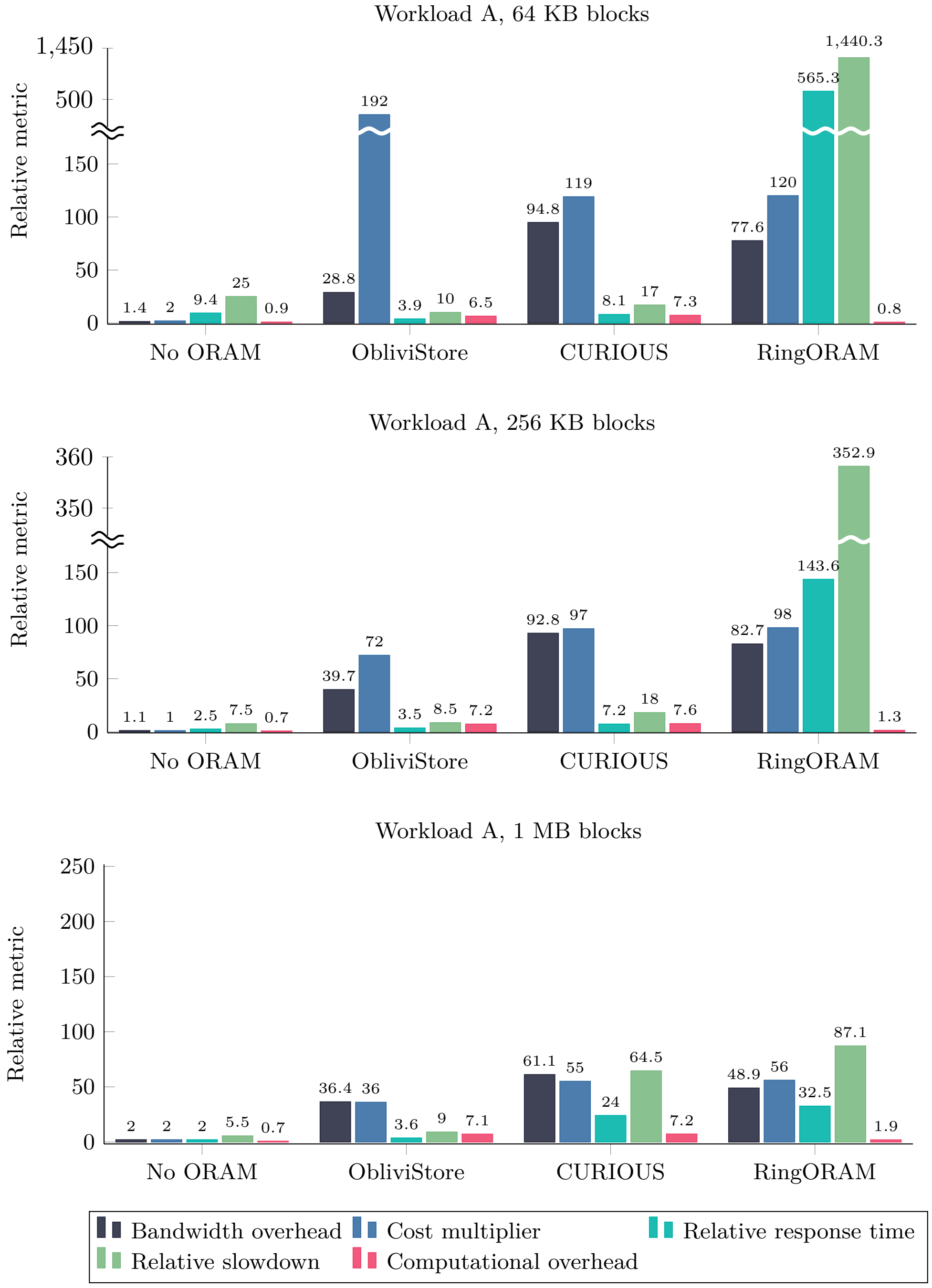


Figure 5.1: Relative metrics for Workload A. Lower is better.

5.1.2 Workload B (Traveling User)

This section presents the results for Workload B (the traveling user workload). For this workload, tests were performed for block sizes of 16 KB, 64 KB and 256 KB.

Baseline (no ORAM, no blocks)	Download operations	76
	Upload operations	24
	MBs downloaded	1.68
	MBs uploaded	7.62
	Runtime (seconds)	5
	Average CPU usage (%)	3.74
	Peak CPU usage (%)	7.6
	Average RAM usage (MB)	97.13
	Peak RAM usage (MB)	144.95
	Read bandwidth (KB/s)	5135.54
	Write bandwidth (KB/s)	2217.25
	Avg. response time read (ms)	111.64
	Avg. response time write (ms)	291.08
	Cost (USD)	~$0.0003
	Bandwidth overhead	1
	Relative response time	1
	Relative slowdown	1
	Cost multiplier	1
	Outsource ratio	1

Table 5.2: Baseline results for Workload B. These results were measured by running the workload without using ORAM and without splitting the files into blocks.

Like in the previous section, Table 5.2 shows the baseline for this workload. The table is divided into two sections. The first section contains data that was collected during the test, while the second section contains relative metrics calculated from the data. Note that the cost is estimated since it was too small to be billable.

The remaining results are summarized in Figure 5.2. The figure shows three bar charts, one for each block size. The charts are divided into four groups, one for each of the three ORAM schemes and one for running the workload without ORAM (but with files being split into blocks). Each group has five bars, one for each of the five overheads that were calculated. These are (from left to right): bandwidth overhead, cost overhead, relative response time, relative slowdown and computational overhead.

As can be seen from the charts, the bandwidth and cost overhead of splitting the files into blocks (without using ORAM) was relatively high for this workload, compared to the other workloads that were tested. The bandwidth overhead increases as the block size increases, with the lowest being 4.5× and the highest being 7.5× higher than the baseline. The cost overhead was 14.8× the baseline when 64 KB blocks were used and 33.5× for the two other block sizes. These overheads were higher than those of any other workload (when no ORAM is used) regardless of block size. A possible explanation for this is the low cost of the baseline. This cost was too low to be included in the IBM Cloud bill and therefore had to be estimated (see Section 6.2). The relative response time and slowdown were less extreme, with overheads ranging from 0.9× to 7× and 4× to 17× respectively. The computational overhead was smaller than or equal to the baseline for all the block sizes that were tested, with the lowest being 0.6× the baseline when using 16 KB blocks.

ObliviStore performed reasonably well for this workload, but incurred high bandwidth and cost overheads. The bandwidth overhead was highest for 16 KB blocks, with 208× more bytes being transferred compared to the baseline, and lowest for 64 KB blocks, with 149.7× more bytes being transferred. The cost overheads ranged from 1509.5× for the largest block size to 20092.9× (the largest overhead across all the workloads and schemes that were tested) for the largest block size. The relative response time and slowdown ranged from 1.4× to 8× and 6× respectively, with both having their minimum at 64 KB blocks and maximum at 256 KB blocks. The computational overhead was smallest for 16 KB blocks, at 8.2× the baseline value and largest for 256 KB blocks, at 9.4× the baseline value.

The overheads of using CURIOUS for this workload were comparable to those of ObliviStore, but with higher bandwidth and lower cost overheads. The bandwidth overheads ranged from 337.5× the baseline value for the smallest block size (16 KB) to 541.5× the baseline value for the largest block size (256 KB). An interesting observation here is that when the bandwidth overhead of CURIOUS was at its lowest (with 16 KB blocks), the bandwidth overhead of ObliviStore was at its highest. The added cost of using CURIOUS for this workload was high, but still lower than that of ObliviStore for all the block sizes that were tested. The cost overhead was smallest for 256 KB blocks, at 1107× the baseline cost, and largest for 16 KB blocks, at 2650× the baseline cost. The relative response time and slowdown ranged from 2.1× to 4.1× and 10× to 19× the baseline value respectively. These values were lower than those of RingORAM and, for all block sizes except 64 KB, ObliviStore. The computational overhead ranged from 9.2× to 11.1× the baseline value. This was the highest of all the schemes for this workload.

RingORAM had high relative response times and slowdowns for this workload, but had the lowest cost of all the ORAM schemes. The bandwidth overhead of RingORAM ranged from 273.8× to 362.9× the baseline, with 64 KB being the bandwidth-optimal block size. The cost overhead also had its minimum for 64 KB blocks, at 805.1× the baseline, with the maximum being at 2113.3× the baseline for 16 KB blocks. The relative response time and slowdown were highest for 16 KB blocks, at 386.1× and 1642.2× the baseline, and decrease as the block size increases. The lowest values for both overheads were 32.1× and 158.4× for 256 KB blocks. The computational overhead of using RingORAM was low compared to the two other schemes, ranging from 0.8× to 2.5× the baseline value. When using 16 KB blocks, the computational overhead of RingORAM was only slighly higher than when no ORAM was used (0.8× vs 0.6×).

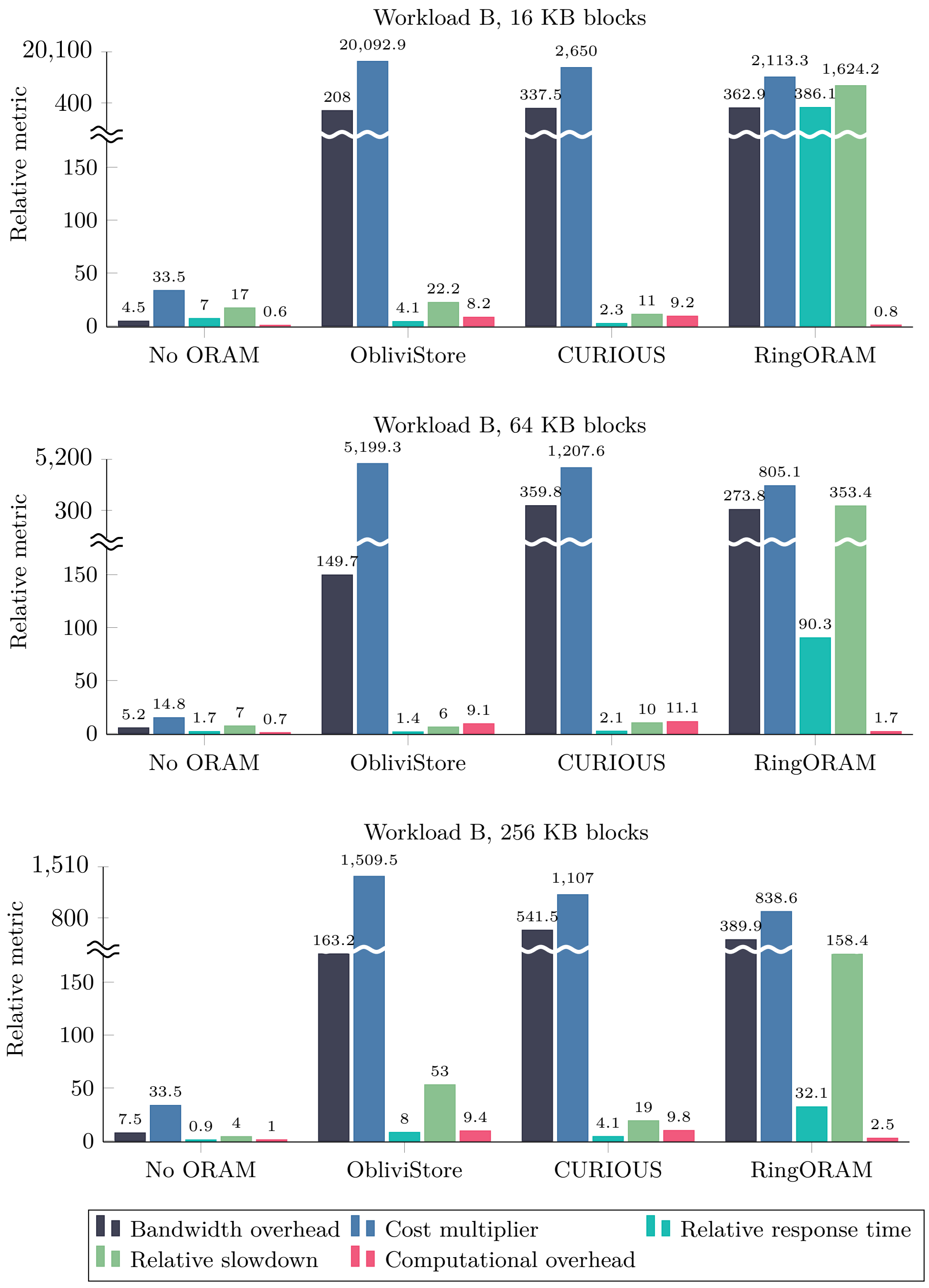


Figure 5.2: Relative metrics for Workload B. Lower is better.

5.1.3 Workload C (Email Backup)

This section presents the results for Workload C (the email backup workload). For this workload, tests were performed for block sizes of 1 MB, 4 MB and 16 MB.

Baseline (no ORAM, no blocks)	Download operations	11
	Upload operations	99
	MBs downloaded	168
	MBs uploaded	1684
	Runtime (seconds)	70
	Average CPU usage (%)	3.89
	Peak CPU usage (%)	5.8
	Average RAM usage (MB)	85.69
	Peak RAM usage (MB)	140.2
	Read bandwidth (KB/s)	1976.4
	Write bandwidth (KB/s)	24312.9
	Avg. response time read (ms)	468
	Avg. response time write (ms)	688.14
	Cost (USD)	$0.02
	Bandwidth overhead	1
	Relative response time	1
	Relative slowdown	1
	Cost multiplier	1
	Outsource ratio	1

Table 5.3: Baseline results for Workload C. These results were measured by running the workload without using ORAM and without splitting the files into blocks.

Like in the previous sections, Table 5.3 shows the baseline for this workload. The table is divided into two sections. The first section contains data that was collected during the test, while the second section contains relative metrics calculated from the data.

The remaining results are summarized in Figure 5.3. The figure shows three bar charts, one for each block size. The charts are divided into four groups, one for each of the three ORAM schemes and one for running the workload without ORAM (but with files being split into blocks). Each group has five bars, one for each of the five overheads that were calculated. These are (from left to right): bandwidth overhead, cost overhead, relative response time, relative slowdown and computational overhead.

For this workload, the overhead of splitting files into blocks (without using ORAM) was similar to other workloads, but with a higher computational overhead. The bandwidth overhead was 1.1× the baseline for 1 and 4 MB blocks and 1.5× for 16 MB blocks. The cost overhead was 1× the baseline value for all three block sizes, meaning that no extra costs were incurred. This is interesting since Workload C is the only workload where this was the case. The relative response time and slowdown ranged from 4.6× to 5.2× and 4.5× to 7.8× the baseline values respectively. The computational overhead ranged from 3.5× the baseline for 1MB to 2.6× the baseline for 16 MB blocks.

ObliviStore incurred low bandwidth overheads for this workload compared to other workloads. The overheads ranged from 24.5× to 30.7× the baseline, with 24.5× being the lowest bandwidth overhead across all the ORAM schemes and workloads that were tested. The cost overheads ranged from 66.5× to 78× the baseline, with 4 MB being the cost-optimal block size. The relative response time and slowdown ranged from 6.6× to 14.2× and 5.1× to 14.4× the baseline value respectively. An interesting observation regarding these metrics is that, for some block sizes, the relative response time was higher than the relative slowdown. Workload C is the only workload where this was the case for ObliviStore. The computational overhead ranged from 15.5× the baseline value for the smallest block size, to 16.9× the baseline for the largest block size. These overheads were higher than for any other workload.

The overheads of CURIOUS were comparable, but slightly higher, than those of ObliviStore for all metrics apart from cost. The bandwidth overheads ranged from 43.4× to 47.1× the baseline value, higher than both ObliviStore and RingORAM for all the block sizes that were tested. The cost overhead ranged from 176.5× the baseline for the smallest block size, to 190.5× the baseline for the largest block size. These overheads were larger than those of ObliviStore for all block sizes, and larger than RingORAM for 4 and 16 MB blocks. The relative response time and slowdown ranged from 13.8× to 21.9× and 12.1× to 22.2× the baseline respectively, with the overheads being smallest for 1 MB blocks and largest for 4 MB blocks. The computational overhead of CURIOUS was lower than that of ObliviStore for most block sizes, ranging from 15.1× to 16.6× the baseline value.

RingORAM incurred high overheads for response time and slowdown, with the remaining metrics being similar to the other schemes. The bandwidth overheads ranged from 25.8× the baseline for the largest block size to 42.6× the baseline for the smallest block size. These were lower than those of CURIOUS for all block sizes and lower than ObliviStore for 16 MB blocks. The cost overheads ranged from 130× to 212× the baseline. This was higher than ObliviStore, but lower than CURIOUS for most block sizes. The relative response time and slowdown ranged from 78.7× to 206.6× and 34.9× to 278.6× the baseline respectively. An interesting observation here is that the response time was lower than the slowdown for 1 MB blocks and higher for the two remaining block sizes. This behavior is the opposite of the other two schemes. The computational overhead of RingORAM ranged from 5× to 10.9× the baseline value. This was lower than the two other ORAM schemes for all block sizes.

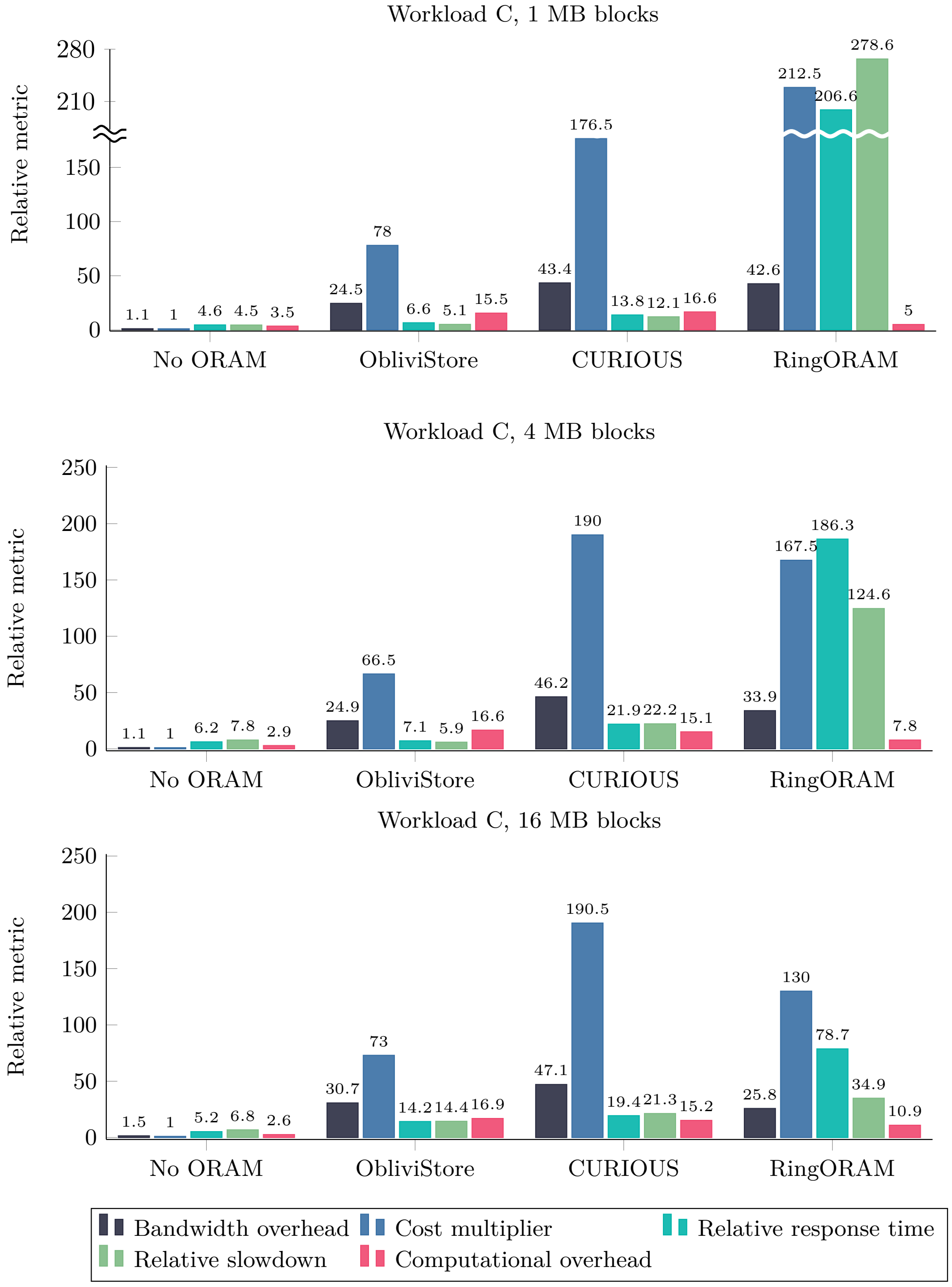


Figure 5.3: Relative metrics for Workload C. Lower is better.

5.1.4 Workload D (Code Distribution)

This section presents the results for Workload D (the code distribution workload). For this workload, tests were performed for block sizes of 16 KB, 64 KB and 256 KB.

Baseline (no ORAM, no blocks)	Download operations	73
	Upload operations	37
	MBs downloaded	17.79
	MBs uploaded	2.36
	Runtime (seconds)	5
	Average CPU usage (%)	2.42
	Peak CPU usage (%)	6.3
	Average RAM usage (MB)	71.89
	Peak RAM usage (MB)	116.1
	Read bandwidth (KB/s)	9328.62
	Write bandwidth (KB/s)	69.78
	Avg. response time read (ms)	109.05
	Avg. response time write (ms)	234.55
	Cost (USD)	~$0.002
	Bandwidth overhead	1
	Relative response time	1
	Relative slowdown	1
	Cost multiplier	1
	Outsource ratio	1

Table 5.4: Baseline results for Workload D. These results were measured by running the workload without using ORAM and without splitting the files into blocks.

Like in the previous sections, Table 5.4 shows the baseline for this workload. The table is divided into two sections. The first section contains data that was collected during the test, while the second section contains relative metrics calculated from the data. Note that the cost is estimated since it was too small to be billable.

The remaining results are summarized in Figure 5.4. The figure shows three bar charts, one for each block size. The charts are divided into four groups, one for each of the three ORAM schemes and one for running the workload without ORAM (but with files being split into blocks). Each group has five bars, one for each of the five overheads that were calculated. These are (from left to right): bandwidth overhead, cost overhead, relative response time, relative slowdown and computational overhead.

The overhead of using blocks (without ORAM) was low compared to the three previous workloads. The bandwidth and cost overheads ranged from 1.1× to 2.5× and 1.3× to 2.1× respectively. The relative response time was 0.8× the baseline for 64 and 256 KB blocks, and 1.8× the baseline for 16 KB blocks. An interesting observation here is that the response time was occasionally lower than the baseline. This was the case only for Workload B and this workload. The relative slowdown ranged from 3× the baseline for the largest block size, to 9× the baseline for the smallest block size. The computational overhead was relatively low, ranging from 0.5× to 1.4× the baseline value.

For this workload, ObliviStore had low overheads for response time and slowdown for most block sizes, but the cost overhead was larger than the two other schemes. The bandwidth overhead ranged from 37.6× to 45.4×. These overheads were lower than the two other schemes for all the block sizes that were tested. The cost overhead ranged from 236.2× the baseline for the largest block size to 3205.9× the baseline for the smallest block size. The relative response time and slowdown ranged from 0.9× to 31.3× and 4× to 52× the baseline respectively. Both values were comparatively low for 16 and 64 KB blocks and significantly higher for 256 KB blocks. The computational overhead ranged from 0.2× to 11.6× times the baseline value. An interesting observation regarding this is that for 64 KB blocks, the computational overhead of using ObliviStore was lower than using no ORAM at all (0.2× vs 0.5× the baseline).

CURIOUS had both high and low overheads for this workload, depending on the block size. The bandwidth overhead ranged from 51.6× the baseline for small blocks, to 120.2× the baseline for large blocks. The cost overhead decreased as the block size increased, starting at 641.2× the baseline for 16 KB blocks and reaching its minimum at 118.1× the baseline for 256 KB blocks. The response time and slowdown ranged from 0.7× to 54.1× and 4× to 340.2× the baseline respectively. Two interesting observations can be made regarding these metrics. The first being their large range, which was unique to this workload, and the second being that they were larger than the overheads of RingORAM for 64 KB blocks. The computational overhead ranged from 10.6× to 10.8×. This range was smaller than that of the two other schemes.

RingORAM had the lowest cost and computational overheads for this workload. Otherwise the overheads were comparable to the two other schemes for most block sizes. The bandwidth overhead ranged from 43.8× to 119.2× the baseline. This overhead was higher than that of ObliviStore, but lower than CURIOUS for 64 and 256 KB blocks. The cost overhead ranged from 78.7× to 219.3× the baseline cost, with 64 KB being the cost-optimal block size. The relative response time and slowdown were high for 16 KB blocks, at 351.2× and 736.6× the baseline values, and comparatively lower for 64 and 256 KB blocks, ranging from 25.7× to 25.9× and 101× to 104.2× the baseline respectively. The computational overhead, ranging from 0.9× to 2.5× the baseline, was lower than the other schemes except for ObliviStore with 64 KB blocks (which had a computational overhead of only 0.2× the baseline).

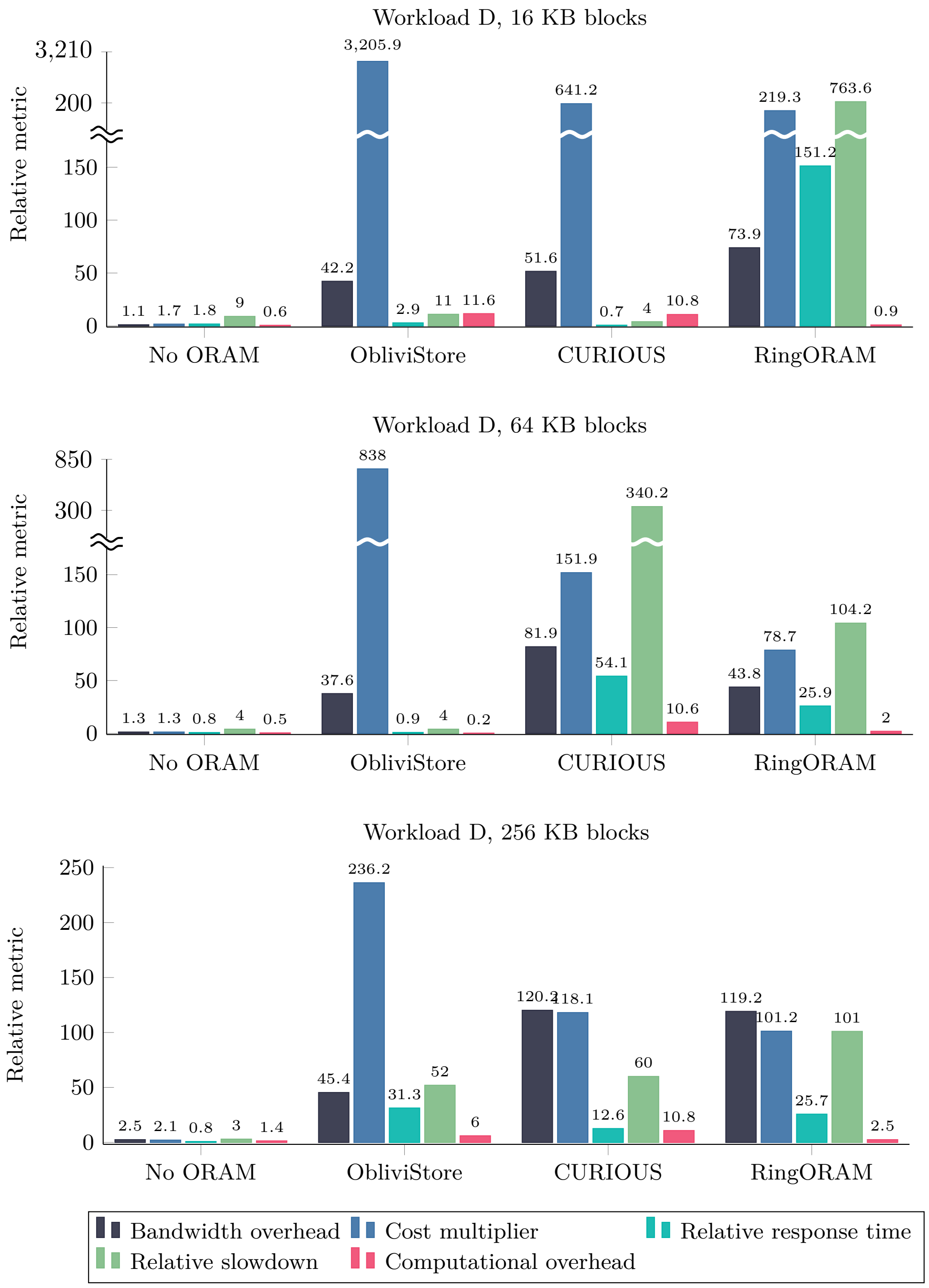


Figure 5.4: Relative metrics for Workload D. Lower is better.

5.1.5 Workload E (E-Government)

This section presents the results for Workload E (the E-Government workload). For this workload, tests were performed for block sizes of 256 KB, 1 MB and 4 MB.

Baseline (no ORAM, no blocks)	Download operations	58
	Upload operations	52
	MBs downloaded	196.64
	MBs uploaded	265.09
	Runtime (seconds)	47
	Average CPU usage (%)	1.99
	Peak CPU usage (%)	15.8
	Average RAM usage (MB)	118.15
	Peak RAM usage (MB)	145.13
	Read bandwidth (KB/s)	7016.53
	Write bandwidth (KB/s)	35180.69
	Avg. response time read (ms)	202.69
	Avg. response time write (ms)	762.4
	Cost (USD)	$0.02
	Bandwidth overhead	1
	Relative response time	1
	Relative slowdown	1
	Cost multiplier	1
	Outsource ratio	1

Table 5.5: Baseline results for Workload E. These results were measured by running the workload without using ORAM and without splitting the files into blocks.

Like in the previous sections, Table 5.5 shows the baseline for this workload. The table is divided into two sections. The first section contains data that was collected during the test, while the second section contains relative metrics calculated from the data.

The remaining results are summarized in Figure 5.5. The figure shows three bar charts, one for each block size. The charts are divided into four groups, one for each of the three ORAM schemes and one for running the workload without ORAM (but with files being split into blocks). Each group has five bars, one for each of the five overheads that were calculated. These are (from left to right): bandwidth overhead, cost overhead, relative response time, relative slowdown and computational overhead.

For this workload the overhead of splitting files into blocks was low for all the considered metrics, compared to the previous workloads. The bandwidth overhead ranged from 1.1× to 1.5× the baseline. The cost overhead was 1.5× the baseline for 256 KB and 4 MB blocks, and 1× the baseline for 1 MB blocks. The relative response time and slowdown ranged from 1.2× to 3.5× and 1.4× to 3.6×, with both overheads decreasing as the block size increased. The computational overhead was 0.5× the baseline for 256 KB blocks and 0.8× the baseline for 4 MB blocks. An interesting observation about this metric is that the computational overhead was smaller than the baseline for all three blocks sizes. This was only the case for this workload and Workload A.

For this workload, ObliviStore had lower overheads than the other ORAM schemes for all metrics except computational overhead. The bandwidth overhead ranged from 26.7× to 34.6× the baseline, with the least bandwidth being consumed when 256 KB blocks were used. The cost overhead ranged from 45.45× the baseline for 256 KB blocks to 24.5× the baseline for 4 MB blocks. The relative response time and slowdown ranged from 2.4× to 3.6× and 2.9× to 4.4× their baseline values respectively. Interestingly, these overheads were lower for ObliviStore than for no ORAM when 256 KB blocks were used. The computational overhead ranged from 5.5× the baseline for the smallest block size, to 6× the baseline for the largest block size. Overall, these were the lowest computational overheads for ObliviStore across all five workloads (barring Workload D with 64 KB blocks).

CURIOUS had high bandwidth and cost overheads for this workload, with the remaining metrics being similar or slightly higher than those of ObliviStore. The bandwidth overheads ranged from 45.4× to 59.3× the baseline, the highest of all the ORAM schemes for this workload. The cost overhead ranged from 46.5× to 63× the baseline cost. An interesting observation to make here is that the cost overhead decreased when going from 256 KB to 1 MB blocks, but then increased again when going from 1 MB to 4 MB blocks. This was not the case for the two other ORAM schemes. The relative response time and slowdown ranged from 5.5× to 12.9× and 6.2× to 21.3× their baseline values respectively. The computational cost was lower for this workload than for any of the other workloads, ranging from 5.1× to 5.7× the baseline value. These values were close to those of ObliviStore for the same block sizes.

RingORAM incurred overall lower overheads for this workload compared to the other workloads. The bandwidth overhead ranged from 38.4×, for the largest block size, to 51.7× the baseline value for the smallest block size. The cost overhead was higher than the two other ORAM schemes for 256 KB and 1 MB blocks, with 70× and 53× the baseline cost, but lower than CURIOUS for 4 MB blocks with 48.5× the baseline value. The relative response time ranged from 133.2× to 16.4× the baseline. Similarly, the relative slowdown ranged from 113.2× the baseline for 256 KB blocks to 18.7× the baseline for 4 MB blocks. An interesting observation here is that the relative response time was larger than the slowdown for 256 KB blocks, but smaller for the two larger block sizes. The computational overhead of using RingORAM for this workload was comparatively small, ranging from 1× the baseline for the smallest block size, to 2.1× the baseline for the largest block size.

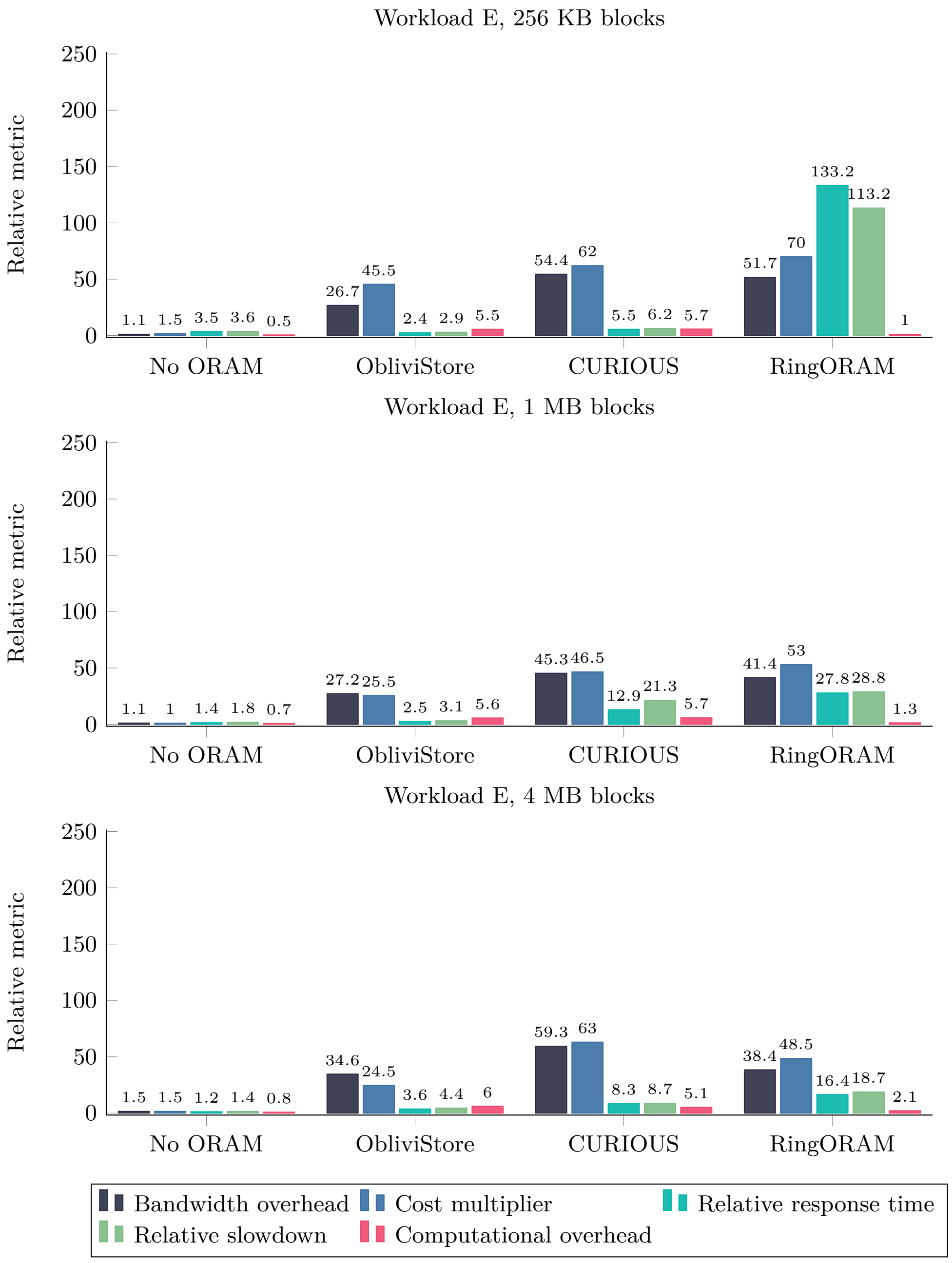


Figure 5.5: Relative metrics for Workload E. Lower is better.

5.2 Real workloads

In this section, the results of running the real-world workloads are presented. These workloads are based on the results of the code distribution workload (Workload D) and consist of four steps (described in Section 4.3.2). For each of these operations, the time required to complete them was measured. The workloads were tested in both a public- and private-cloud scenario, as described in Section 4.4.

		Elapsed Time
Baseline (No ORAM)	Pushing a repo	19.7 s
	Pulling a repo	8 m 14.2 s
	Pushing a commit	2.4 s
	Pulling a commit	26.7 s
	Cost	$0.16

Table 5.6: Results of running the four steps of the real-world code distribution workload without using ORAM in a public-cloud scenario. Costs are in USD. Elapsed time is given in minutes (m) and seconds (s).

Table 5.6 presents the baseline for this workload. These results were obtained by running tests against the cloud storage directly, without using the ORAM proxy. The table contains the time required to complete each of the four steps as well as the combined cost of all steps (taken from the IBM Cloud bill). As can be seen from the table, the different steps took varying amounts of time, with the slowest step being the initial cloning of the repository (taking 8 minutes and 14.2 seconds) and the quickest step being pushing a new commit (taking 2.4 seconds).

		Elapsed Time	Relative Slowdown
CURIOUS (16 KB blocks)	Pushing a repo	33.4 s	1.70
	Pulling a repo	82 m 38.2 s	10
	Pushing a commit	2.3 s	0.96
	Pulling a commit	35.1 s	1.32
	Cost	$17.60	
ObliviStore (64 KB blocks)	Pushing a repo	24.9 s	0.79
	Pulling a repo	94 m 4.7 s	11.42
	Pushing a commit	1.4 s	0.58
	Pulling a commit	34.8 s	1.30
	Cost	$23.39	

Table 5.7: Results of running the four steps of the real-world code distribution workload with ORAM in a public-cloud scenario. Costs are in USD.

The results of running the workloads in a public-cloud scenario are presented in Table 5.7. The table is divided into two sections, one for each of the schemes that were tested. For each scheme, the time it took to complete each step is given. The total cost is also included.

As can be seen from the results, the time required to complete the different steps varied greatly. Pushing a new repository to the server took 33.4 seconds when CURIOUS was used and 24.9 seconds when using ObliviStore. Cloning the same repository took a lot longer, with CURIOUS using 82 minutes and 38.2 seconds and ObliviStore needing as much as 94 minutes and 4.7 seconds to complete the clone. The two last steps favored ObliviStore, with pushing a new commit taking 1.4 seconds (versus to 2.3 seconds for CURIOUS) pulling a commit taking 34.8 seconds (versus 35.1 seconds for CURIOUS). The cost of completing all four steps was $17.6 for CURIOUS and $23.39 for ObliviStore.

		Elapsed Time	Relative Slowdown
CURIOUS (16 KB blocks)	Pushing a repo	51.1 s	2.59
	Pulling a repo	124 m 18.9 s	15.09
	Pushing a commit	2.5 s	1.04
	Pulling a commit	37.1 s	1.39
	Cost	$17.60	
ObliviStore (64 KB blocks)	Pushing a repo	36.8 s	1.87
	Pulling a repo	120 m 1.5 s	14.57
	Pushing a commit	1.2 s	0.50
	Pulling a commit	25.5 s	0.95
	Cost	$23.39	

Table 5.8: Results of running the four steps of the real-world code distribution workload with ORAM in a private-cloud scenario. Costs are in USD.

The results of running the same workloads in a private-cloud scenario are presented in Table 5.8. The table is divided into two sections, one for each of the schemes that were tested. For each scheme, the time it took to complete each step is given. The total cost (taken from the IBM Cloud bill) is also included.

These results are comparable to those of the public-cloud scenario, but with most of the steps taking more time. The exception to this is pushing and pulling new commits when using ObliviStore. In these cases the steps took 1.2 and 25.5 seconds, compared to 1.4 and 34.8 seconds in the public-cloud case. The costs were the same as in the public-cloud case for both schemes.

Chapter 6
Discussion

In this chapter, the results from Chapter 5 are discussed. First, the results are evaluated in the context of the research questions from Chapter 1 and then in the broader context of the field of Oblivious RAM. The validity of the methodology and the results is then discussed and potential shortcomings are highlighted. Finally, the other contributions, apart from the results, made by this thesis are presented.

For reference, the research questions of the thesis are restated here. The main research question is *Are current ORAM schemes practical in a cloud setting?*. This is a very broad question and it has therefore been split into five more narrow research questions:

- Which metrics determine the practicality of ORAM in a cloud setting?
- Which types of workloads are found in practical cloud applications?
- How do current ORAM schemes perform under realistic workloads?
- What are the costs associated with using ORAM with cloud storage?
- Are there other factors that need to be considered when using ORAM in practice?

6.1 Evaluation of Results

As stated above, the overarching research question of this thesis is: *Are current ORAM schemes practical in a cloud setting?* The answer to this question will depend strongly on what is considered practical, with the requirements differing greatly from user to user. It is therefore not possible to give a definite answer to the question. Instead, this section focuses on the specific use cases that each of the workloads were based on and subjectively evaluates their practicality. The aim of this evaluation is to provide some insight regarding the last three research questions.

The two first workloads (Workload A and B) are based on an individual user that uses ORAM to secure their personal files. For this evaluation, it is assumed that cost is the most important factor for a home user. The lowest measured cost across both workloads was 36× higher than the baseline cost ($0.36 vs $0.01). At first glance, this would seem like an unacceptably high cost overhead for anything but storing a few small files, however this cost is highly dependent on the pricing model used by the cloud provider. Personal cloud storage providers generally charge a flat fee per month or charge per GB stored, with bandwidth and requests being free of charge. In such a model, the most important metrics are the external storage overhead (i.e how much extra storage the ORAM scheme requires), which determines the cost, and the response time and slowdown, which determine usability.

The external storage overhead was not directly studied, but the overall external storage used by each of the schemes was recorded and is available in Appendix C. This data shows that CURIOUS used the least external storage. An ORAM size of 5GB was used for all tests, meaning that the best case external storage overhead for CURIOUS works out to be about 2.2× and 1.6× for Workload A and B respectively. With these overheads, a user is able to store approximately 6.82 and 9.38 GBs on Google Drive, which provides individuals with up to 15 GBs of storage for free. This is within the realm of practicality.

The response times and slowdowns affect the usability by reducing the transfer rate seen by the user when uploading or downloading files from the ORAM. The response time affects single file transfers, while the slowdown affects a series of transfers. Assuming that the user requests single files more often than multiple files, the response time has the biggest effect on practicality. The lowest response time overhead for the two workloads was 3.5× and 1.36× higher than the baseline value respectively. These overheads are also within the realm of practicality, however to achieve these overheads a higher external storage overhead is required.

The conclusion is therefore that ORAM could be practical for a home user as long as some tradeoffs can be made. More external storage and lower transfer rates will be required, with the user having to accept high overheads for one of the two. Ultimately the question of practicality depends on the users acceptable security level. Simply encrypting the files before uploading them to the cloud will likely give an acceptable level of security for most users.

Workloads C, D and E are based on a scenario where an organization (either private or governmental) uses ORAM for different purposes. In the first scenario (Workload C) the organization runs an email server and needs to back up large (10-25 MB) email attachments. For this analysis, it is assumed that these backups are infrequently accessed and deleted after a certain period of time. Furthermore it is assumed that the backups are performed every night and have ample amounts of time to complete. The most important factor determining the practicality of this scenario is therefore the cost. It is assumed that the organization is less cost sensitive than the home user, but the added security still needs to be worth the added costs.

The lowest cost overhead for Workload C was 66.5× the baseline cost. In this best-case scenario, 1684 MB were backed up for a total cost of \$1.33 (compared to the baseline cost of \$0.02). Assuming that the average size of a large attachment is 17.5 MB, the organization would be able to back up approximately 96 attachments for this cost. If we assume that this cost is linearly related to the number of attachments (a shaky assumption at best), the cost per attachment works out to about \$0.014. This calculation does not include the cost of storage over time so the actual amount would be higher. These costs are much higher than a backup solution without ORAM, but not prohibitively expensive. If 1000 large attachments need to be backed up every day, the cost (excluding storage) would be \$420 per month.

It is difficult to draw conclusions about this scenario, due to the potentially large inaccuracies involved in the calculations. However it would seem that backing up email attachments using ORAM could be practical, since this workload can tolerate large bandwidth, response time and other overheads as long as the cost overhead is manageable.

The second organization scenario is based around an organization that uses a version control system to distribute their proprietary SDK. For this scenario both a synthetic workload (Workload D) and a real workload (the Git workload) were tested. A more in-depth analysis of the practicality of this scenario is therefore possible.

The major factors impacting the practicality of this workload is cost (e.g how much does it cost when new customers start using the SDK, how much does it cost to push an update, etc.) and slowdown (e.g how long does it take for developers to push updates and for customers to download them). The results of running the synthetic workload show that a tradeoff has to be made with these two metrics. When one of the metrics has a low overhead, the other metric has a high overhead and vice versa. The lowest cost is achieved with RingORAM (at 78.7× the baseline cost for 64 KB blocks) and the lowest slowdown with ObliviStore (at 4× the baseline for the same block size). CURIOUS lies somewhere in between (with a cost overhead of 118.1× and a slowdown of 60× for 256 KB blocks) as a possible compromise.

For the real workload (the Git workload) this tradeoff was made in favor of a low relative slowdown, at the expense of a higher cost. This is therefore the scenario that is considered for the rest of the evaluation. The workload was only tested with CURIOUS and ObliviStore, due to lack of time to test RingORAM. The tests resulted in very manageable slowdowns. Three of the four operations were able to be completed in less than a minute, with slowdowns ranging from 0.58× to 1.70× the baseline value in the public-cloud case and 0.50× to 2.59× the baseline value in the private-cloud case (see Tables 5.7 and 5.8). The operation requiring the most time was cloning a repository for the first time, with slowdowns ranging from 10× to 15.09× the baseline value. This overhead is significant, but because of the scenario considered for this workload, the operation in question will not occur very often (only when new customers first start using the SDK).

The costs of running the Git workload were large, ranging from 110× to 146.19× the baseline cost. Unfortunately, the cost was not measured for each operation, only for the workload as a whole, so it is not known which steps contributed most to the cost. Without knowing this, it is difficult to evaluate whether the costs are prohibitively high or not.

The conclusion for this scenario is therefore that it can be practical if high costs are acceptable. There is not enough information to make any conclusions for other cases (e.g when choosing to trade low slowdowns for low costs).

The final workload (Workload E) is based on a scenario where ORAM is used in an E-Goverment system. This workload differs from the others in that it tries to model a scenario where traffic is *bursty*. This means that the ORAM has to handle a lot of reads for a certain period of time and then a lot of writes for another. It is assumed that the E-Government system in question is sufficiently important that costs are less of a concern. Because of this, the important metrics for this scenario are response time (i.e being able to quickly respond to requests during peak periods) and computational overhead (i.e being able to handle a large amount of users without overloading the server).

From the results shown in Figure 5.5 we see that no single ORAM scheme / block size combination minimizes both of these metrics. RingORAM incurs the lowest computational overhead, while ObliviStore incurs the lowest relative response time). Selecting the scheme with the lowest computational overhead gives a response time that is 113.2× higher than the baseline (averaging around 64 seconds per request). If one instead optimizes for response time, a computational overhead of 5.5× the baseline value is incurred (peaking at 87.1% CPU usage). There is no good compromise for the schemes and block sizes that were tested, with both ObliviStore and CURIOUS having peak CPU usages of 80% or more and RingORAM having average response times on the scale of minutes.

The conclusion is therefore that this scenario is impractical for the schemes and block sizes that were tested. The relative response times and computational overheads are simply too high to support a bursty traffic pattern with many users requesting service at the same time. Another concern is that none of the schemes that were tested are Oblivious Paralell RAMs (OPRAMs) and are therefore unable to handle multiple application-level requests at the same time. This would be very limiting in a real world deployment, since this scenario needs to support many concurrent users.

To summarize, most of the scenarios that were tested are impractical in the pricing model of the cloud storage service they were tested with (IBM Cloud). Some of the scenarios could be practical if high costs are acceptable or if a different pricing model is available. These scenarios are generally ones with more writes than reads. The reason for this is that writes are more expensive than reads in most pricing models (see Table 4.5). When using ORAM, the same number of read and write requests need to be performed to serve an application-level request, regardless of its type. This means that extra cost of using ORAM is lower if the baseline has many writes. Another factor affecting the practicality is a scenario's tolerance for lower performance. Backup scenarios like the one considered in Workload C can tolerate longer running times and lower troughput, while realtime applications like that of Workload E can not.

6.2 Limitations

This section points out potential limitations and sources of error in the experiments that were conducted. First, high level limitations are addressed, followed by limtations of the implementations that were used and the tests that were carried out.

A potential limitation of the methodology that was used is that not all research questions were given equal treatment. The main focus was on the tests that were run and therefore on the three last research questions. The first research question *(Which metrics determine the practicality of ORAM in a cloud setting?)* deals with the metrics used when evaluating and comparing ORAM schemes. Most of the metrics used in this thesis were selected based on the results of Bindschaedler et al. [BNP+15], with the exception being the computational overhead metric, which was selected based on intuition gained while implementing the ORAM proxy. Pros of using these metrics are that they are easy to measure/calculate and they are understandable for people outside the ORAM community. The cons of these metrics is that they are not established in the literature, meaning that results are not easily comparable.

The second question (*Which types of workloads are found in practical cloud applications?*) is related to the workloads that were used when running tests. For the results to be applicable to the real world, workloads should mirror real cloud applications as much as possible. This is an area that has not been given sufficient treatment in previous studies (in this case the study by Bindschaedler et al. [BNP+15]). The original plan for this thesis was to use only real-world workloads, generated by representative cloud-applications. This proved to be difficult due to the lack of compatible and freely available cloud applications. The majority of workloads were instead based on a few realistic scenarios, with the parameters of the workloads being based on either previous academic surveys (Workload A-C and E) or data collected from publicly available sources (Workload D). Only a small number of workloads were generated using a real cloud application (Git). The details of how each workload was selected is given in Section 4.3. The pros of this approach is that more data points could be collected, since the manual labour required to configure real applications for different scenarios and block sizes was avoided. The cons of this approach is that results could be less applicable to the real world.

Regarding the implementation and test setup, there are several potential limitations. The implementations of the ORAM schemes that were tested were not verified to be correct and no guarantees can be made about their performance. This means that better implementations could be possible, thus giving better results than what was found in this thesis. It also means that the implementations could have errors that might affect the results and in turn, the conclusions drawn in the previous section.

Asynchronicity is a big factor affecting the performance of ORAM schemes in a cloud scenario (as pointed out by Bindschaedler et al. [BNP$^+$15]). Two of the schemes that were tested (ObliviStore and CURIOUS) are asynchronous. These schemes should ideally be compared to a baseline that is also asynchronous, both with and without blocks. This was not fully possible in the tests that were conducted due to a bug in the ORAM proxy. Because of the bug, tests without ORAM (but with blocks) could only be performed synchronously. This means that of the two baselines, only one of them is directly comparable to the two asynchronous schemes.

The effect of this can be seen from the results in Chapter 5. For example, the response times and slowdowns of both ObliviStore and CURIOUS are smaller than the baseline with blocks for Workload A when a block size of 64 KB is used (see Figure 5.1). Similar results can be found for the other workloads.

Another factor affecting the accuracy of the results is the "resolution" of the IBM Cloud billing system. The smallest billable amount is \$0.01, meaning that smaller amounts were not be included in the bills. Because of this, the baseline costs for Workload B and D had to be estimated from the bandwidth / number of requests. This estimation introduces some unknown uncertainty into both the baseline costs and the cost overheads for these workloads.

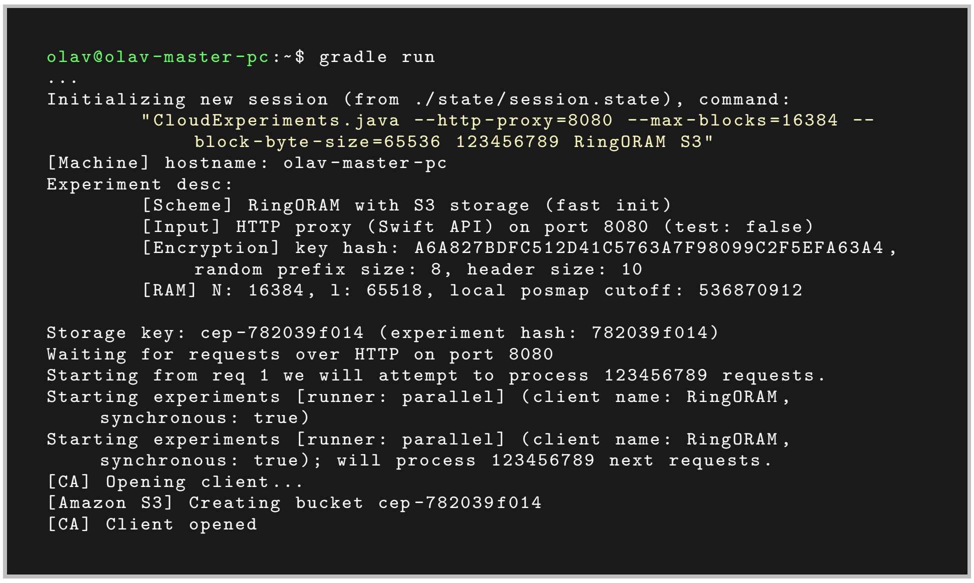
6.3 Relation to Previous Work

This section considers the findings and implications of this thesis in a broader context. As mentioned in Chapter 1, the study by Bindschaedler et al. [BNP+15] is the only previous work to compare the performance of ORAM schemes in a cloud scenario. The results of this thesis are largely in line with those of Bindschaedler et al. [BNP+15]. As such, this thesis can be seen as a continuation of their work, but with more realistic workloads and updated to use newer schemes (RingORAM).

6.4 Other Contributions

In addition to the results from Chapter 5 and the insight gained from them, other contributions to the field of ORAM have been made while writing this thesis. These contributions are in the form of tools that can be used in future studies.

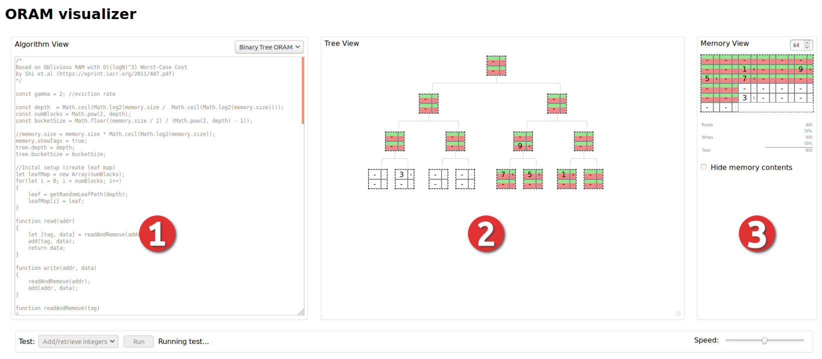
The first of these tools is the ORAM proxy. This tool allows running cloud applications that supports the Swift API with any of the three ORAM schemes tested in this thesis. The code for the ORAM proxy is based on the codebase released by Bindschaedler et al. [BNP+15], but with a number of modifications. The design of the ORAM proxy is modular, meaning that it is easy to add support for both new ORAM schemes and cloud storage APIs. The ORAM proxy, include its source code, has been made publicly available at `https://github.com/olav-st/oram-proxy`.



```
olav@olav-master-pc:~$ gradle run
...
Initializing new session (from ./state/session.state), command:
        "CloudExperiments.java --http-proxy=8080 --max-blocks=16384 --
            block-byte-size=65536 123456789 RingORAM S3"
[Machine] hostname: olav-master-pc
Experiment desc:
        [Scheme] RingORAM with S3 storage (fast init)
        [Input] HTTP proxy (Swift API) on port 8080 (test: false)
        [Encryption] key hash: A6A827BDFC512D41C5763A7F98099C2F5EFA63A4,
            random prefix size: 8, header size: 10
        [RAM] N: 16384, l: 65518, local posmap cutoff: 536870912

Storage key: cep-782039f014 (experiment hash: 782039f014)
Waiting for requests over HTTP on port 8080
Starting from req 1 we will attempt to process 123456789 requests.
Starting experiments [runner: parallel] (client name: RingORAM,
    synchronous: true)
Starting experiments [runner: parallel] (client name: RingORAM,
    synchronous: true); will process 123456789 next requests.
[CA] Opening client...
[Amazon S3] Creating bucket cep-782039f014
[CA] Client opened
```

Figure 6.1: Example output when running the ORAM proxy with RingORAM as the scheme and Amazon S3 as the backend.



```
/*
Based on Oblivious RAM with O((logN)^3) Worst-Case Cost
by Shi et.al (https://eprint.iacr.org/2011/407.pdf)
*/

const gamma = 2; //eviction rate

const depth  = Math.ceil(Math.log2(memory.size /  Math.ceil(Math.log2(memory.size))));
const numBlocks = Math.pow(2, depth);
const bucketSize = Math.floor((memory.size / 2) / (Math.pow(2, depth) - 1));

//memory.size = memory.size * Math.ceil(Math.log2(memory.size));
memory.showTags = true;
tree.depth = depth;
tree.bucketSize = bucketSize;

//Inital setup (create leaf map)
let leafMap = new Array(numBlocks);
for(let i = 0; i < numBlocks; i++)
{
    leaf = getRandomLeafPath(depth);
    leafMap[i] = leaf;
}

function read(addr)
{
    let [tag, data] = readAndRemove(addr
    add(tag, data);
    return data;
}

function write(addr, data)
{
    readAndRemove(addr);
    add(addr, data);
}

function readAndRemove(tag)
```

Figure 6.2: Screenshot of the ORAM visualizer created while researching for this thesis. The three most important elements, the algorithm view, the logical view and the memory view, are numbered from 1-3.

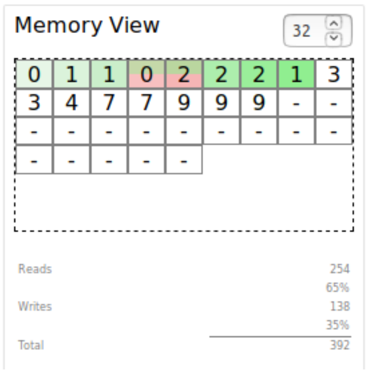
The second tool is designed to help explain ORAM to people that are unfamiliar with the concept. It can be difficult to get an intuitive understanding of how ORAM schemes work. The papers describing them often focus on theoretical proofs, providing only a short and dense explanation of the schemes themselves. Some ORAM schemes have open-source implementations available, but these are usually geared towards specific research projects and the code is normally not well documented[1].

Because of this, a tool was created to illustrate ORAM schemes graphically, called the *ORAM visualizer*. The tool simulates different workloads (e.g. linear search, sorting an array, etc.) on a simulated memory. The workloads can be performed either directly on the simulated memory, or through an ORAM scheme selected by the user. All workloads are small and can be performed in "slow-motion". Slowing down the execution makes it possible to observe the behaviour of the ORAM schemes while the workload is running.

The ORAM visualizer is implemented as a web application. A screenshot of the user interface is shown in Figure 6.2. The user interface is organized into different *views*, numbered 1-3 in the figure. A description of each view is given below:

1. The *Algorithm View* shows the code of the ORAM scheme currently in use. A dropdown menu lets a user select from a number of ORAM schemes. Using a custom scheme is also supported, in case the user wants to provide their own code. The dropdown menu can also be used to turn ORAM off entirely.

[1]An example of this is SEAL-ORAM, a testbed for evaluating ORAM schemes (available at https://github.com/InitialDLab/SEAL-ORAM). Many prominent schemes are implemented, but they are cluttered with verbose encryption code and documentation is lacking.



0	1	1	0	2	2	2	1	3
3	4	7	7	9	9	9	-	-
-	-	-	-	-	-	-	-	-
-	-	-	-	-				

Reads	254
	65%
Writes	138
	35%
Total	392

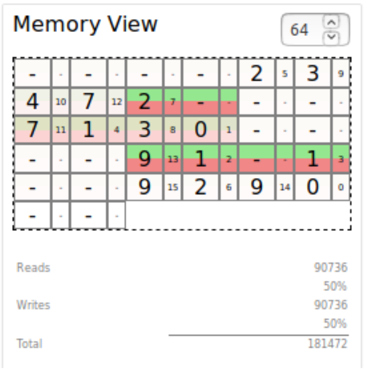
Figure 6.3: Screenshot of the memory view shown by the visualizer when no ORAM scheme is used. Access pattern is clearly visible.

2. The *Logical View* shows the logical structure used by the ORAM scheme, e.g. a binary tree for Tree-Based ORAM schemes (the only logical view currently supported).

3. The *Memory View* shows the simulated memory and some statistics about the access pattern to it. The size of the memory can be adjusted by the user. Hiding of the values stored in the memory can also be turned on, giving the user a view similar to that of an adversary in the models described in Section 1.1.

The Memory View is the most important part of the visualizer and will therefore be explained more thoroughly. Figure 6.3 shows a snapshot of the memory view when no ORAM scheme is used and the workload is bubble sort on a list of 16 integers. Bubble sort works by linearly scanning through the list and swapping pairs of elements if the left element is larger than the right element. This behaviour is reflected in the access pattern to the simulated memory. We can see that the green cells have been scanned by the sorting algorithm, but were not swapped since they were already in order. The red/green cells have recently both been read and written to, which means that they originaly were unordered and the algorithm swapped them.

Figure 6.4 shows the same scenario (bubble sort on a list of 16 integers), but with a Tree-Based ORAM scheme in use. The access patterns no longer reveal what algorithm is running or which elements are being swapped. In fact, every time a cell is read from it is also written to and vice versa. This leads to an even distribution of 50% reads and 50% writes. This comes at a cost of more memory operations and the need to store metadata (illustrated by a small number next to every cell).



Reads	90736
	50%
Writes	90736
	50%
Total	181472

Figure 6.4: Screenshot of the memory view shown by the visualizer when a Tree-Based ORAM scheme is in use. Access pattern is now obscured.

The source code for the visualizer has been made available online at `https://github.com/olav-st/oram-visualizer`. A live version is also available, at `http://folk.ntnu.no/olavsth/oram`. It is hoped that this tool will be a useful resource for people that are new to ORAM.

Chapter 7

Conclusion & Future Work

This chapter summarizes the results of the thesis and suggests some future work. First, the goals of the thesis are restated and the main findings and conclusions are highlighted. Recommendations for future work are then given.

Conclusion

This thesis aimed to study the practicality of current ORAM schemes in a cloud setting. This was done by running tests with both synthetical workloads and workloads generated by a real cloud application (Git). The results (presented in Chapter 5 and discussed in Chapter 6) indicate that the ORAM schemes tested in this thesis incur significant overheads for most workloads and as such are impractical in most scenarios. Cost and slowdown were the major factors hindering practicality.

Despite high overheads, a handful of scenarios where ORAM could be practical were identified. One of these scenarios is a home user using ORAM to securely store their files. This scenario was identified as potentially practical because of the pricing model of personal cloud storage services. The services generally do not charge for upload/download operations or bandwidth, but instead charge a flat monthly fee determined by the amount of storage used. The use of ORAM was found to be more practical under this type of pricing model. Another scenario that was found to be practical was using ORAM to back up email attachments. In this scenario, the average file size is known beforehand and an optimal block size and ORAM scheme can thus be chosen. This scenario can also handle large performance overheads, since the backups rarely need to be accessed. The costs of this scenario were found to be within reasonable limits for a large organization.

To summarize, the results of this thesis are in line with previous results (particularly those of Bindschaedler et al. [BNP$^+$15]), but an improved test setup and more realistic workloads have made it possible to identify new scenarios where ORAM could be practical in a cloud setting.

Future Work

This section suggests some future work based on the results of this thesis. These suggestions include both tests that were not carried out due to resource constraints (time, money, etc.) and ways to address the limitations of this study.

A natural extension of this thesis would be to run tests with a wider range of block sizes. In particular, larger block sizes would be interesting because for certain combinations of ORAM schemes and workloads (e.g RingORAM and Workload E), all of the overheads that were measured decrease as the block size increases. By studying a larger range of block sizes, the optimal block sizes for different schemes could be determined. This would provide more insights into the overheads of using ORAM on the cloud and could potentially be used to estimate the practicality of new ORAM schemes.

A study comparing the results of this thesis to the results of similar tests using DP-ORAMs (see Section 3.11) would also give valuable insight. DP-ORAMs are tunable, allowing a tradeoff between access pattern privacy and performance. Many practical applications do not require perfect privacy and would therefore benefit from the potentially lower overheads of these schemes.

It would also be interesting to study the effects of link quality (packet loss, bandwidth restrictions, etc.) on ORAM performance. Such a study would shed light on the practicality of using ORAM on mobile networks. This would have implications for the envisioned Internet of Things (IoT), where embedded devices are ubiquitous and connected to the Internet. These devices are likely to take advantage of cloud storage due to their limited local storage. Solutions like ORAM can therefore be a useful way to enhance the privacy and security of these devices.

References

[Avr14] M.G. Avram. Advantages and Challenges of Adopting Cloud Computing from an Enterprise Perspective. *Procedia Technology*, 12:529 – 534, 2014. The 7th International Conference Interdisciplinarity in Engineering, INTER-ENG 2013, 10-11 October 2013, Petru Maior University of Tirgu Mures, Romania.

[Bat68] Kenneth E. Batcher. Sorting Networks and Their Applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, pages 307–314, 1968.

[BCP16] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious Parallel RAM and Applications. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography*, pages 175–204, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[BNP$^+$15] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 837–849, 2015.

[CLP14] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 62–81, 2014.

[CP13] Kai-Min Chung and Rafael Pass. A Simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.

[CS14] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014.

[DH11] Hrishikesh Dewan and R. C. Hansdah. A Survey of Cloud Storage Facilities. In *World Congress on Services, SERVICES 2011, Washington, DC, USA, July 4-9, 2011*, pages 224–231, 2011.

[DvDF$^+$16] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography*, pages 145–174, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[Dwo06] Cynthia Dwork. Differential Privacy. In *33rd International Colloquium on Automata, Languages and Programming, part II (ICALP 2006)*, volume 4052 of *Lecture Notes in Computer Science*, pages 1–12. Springer Verlag, July 2006.

[GGH$^+$13a] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using It Efficiently for Secure Computation. In Emiliano De Cristofaro and Matthew Wright, editors, *Privacy Enhancing Technologies*, pages 1–18, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GGH$^+$13b] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, pages 1–18, 2013.

[GL18] Kimia Ghaffari and Mohammad Lagzian. Exploring users' experiences of using personal cloud storage services: a phenomenological study. *Behaviour & Information Technology*, 37(3):295–309, 2018.

[GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, pages 13–24, 2012.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[Gol87] Oded Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194, 1987.

[Goo10] Michael T. Goodrich. Randomized Shellsort: A Simple Oblivious Sorting Algorithm. *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan 2010.

[HHNZ19] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[Hol16] Kjell Jørgen Hole. *Toward an Anti-fragile e-Government System*, pages 57–65. Springer International Publishing, Cham, 2016.

[HRAAN12] Mohammed Hawa, Jamal S. Rahhal, and Dia I. Abu-Al-Nadi. File size models for shared content over the BitTorrent Peer-to-Peer network. *Peer-to-Peer Networking and Applications*, 5(3):279–291, Sep 2012.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[KHK10] Josef Kanizo, David Hay, and Isaac Keslassy. Maximum Bipartite Matching Size And Application to Cuckoo Hashing. *CoRR*, abs/1007.1946, 2010.

[Köh15] Jens Köhler. *Tunable Security for Deployable Data Outsourcing*. PhD thesis, Karlsruhe Institute of Technology, 2015.

[Lam14] Mark Lamourine. OpenStack. *;login: The USENIX Magazine*, 39(3):18–20, June 2014.

[LHFY13] S. Liu, X. Huang, H. Fu, and G. Yang. Understanding Data Characteristics and Access Patterns in a Cloud Storage System. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 327–334, May 2013.

[MG11] Peter M. Mell and Timothy Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report, Gaithersburg, MD, United States, 2011.

[MLS+13] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 311–324, New York, NY, USA, 2013. ACM.

[Ost90] Rafail Ostrovsky. Efficient Computation on Oblivious RAMs. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 514–523, 1990.

[PR10] Benny Pinkas and Tzachy Reinman. Oblivious RAM Revisited. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 502–519, 2010.

[RFK+15] Ling Ren, Christopher W. 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, Washington, D.C., USA, August 12-14, 2015.*, pages 415–430, 2015.

[RFK+18] L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas. Design and Implementation of the Ascend Secure Processor. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2018.

[RGR18] David Reinse, John Gantz, and John Rydning. White Paper: The Digitization of the World - From Edge to Core. Technical Report US44413318, International Data Corporation, Framingham, Massachusetts, November 2018.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with O((log N)^3) Worst-Case Cost. *IACR Cryptology ePrint Archive*, 2011:407, 2011.

[SS13] Emil Stefanov and Elaine Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 253–267, 2013.

[SSS11] Emil Stefanov, Elaine Shi, and Dawn Song. Towards Practical Oblivious RAM. *CoRR*, abs/1106.3652, 2011.

[SvDS$^+$13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310, 2013.

[Tee15] Paul Teeuwen. Evolution of oblivious RAM schemes. Master's thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2015.

[Wan14] Yaguang Wang. *COSBench User Guide.* Intel, Nov 2014.

[WCM16] Sameer Wagh, Paul Cuff, and Prateek Mittal. Root ORAM: A Tunable Differentially Private Oblivious RAM. *CoRR*, abs/1601.03378, 2016.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 850–861, 2015.

[WHC$^+$14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 191–202, 2014.

[WS08] Peter Williams and Radu Sion. Usable PIR. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

[WST12] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A Parallel Oblivious File System. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 977–988, New York, NY, USA, 2012. ACM.

[ZCW$^+$12] Qing Zheng, Haopeng Chen, Yaguang Wang, Jiangang Duan, and Zhiteng Huang. COSBench: A Benchmark Tool for Cloud Object Storage Services. In *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 998–999, 2012.

Appendix A
Workload Files

This appendix gives a more detailed description of the workloads presented in Section 4.3. First, the format of the workload files is described, followed by a brief explanation of each workload. The full workload specification, in XML-format, is also provided.

COSBench workloads are represented by XML-files. The format of these files is briefly described in this section. A more detailed description can be found in the COSBench User Guide [Wan14]. A workload has (at least) five stages: `init`, `prepare`, `main`, `cleanup` and `dispose`. The purpose of each stage is explained below:

– The `init` stage creates the container(s) that will be used to hold objects for this workload.

– The `prepare` stage uploads the objects that will later be used to test download performance in the main stage.

– The `normal` stage is the main stage of the workload. In this stage the objects from the previous stage are downloaded and new objects are uploaded.

– The `cleanup` stage deletes the objects that were uploaded by the other stages.

– Finally, the `dispose` stage deletes the container(s) that were used while running the workload.

The `normal` stage is a general-purpose stage and can appear more than once in a workload. An example of this can be seen in the XML-file for Workload E. This workload has 5 `normal` stages, with the stages alternates between a high proportion of reads and a high proportion of writes. This is an attempt at modeling a scenario with bursty traffic. For the remaining workloads (A-D) a single `normal` stage is sufficient.

A.1 Workload A (Home User)

```
<?xml version="1.0" encoding="UTF-8"?>
<workload name="WorkloadA" description="Home User" config="">

    <auth type="swauth" config="timeout=36000000;username=test:tester;
         password=testing;auth_url=http://olav-openstack-test-server:8080/
         auth/v1.0"/>
    <storage type="swift" config="timeout=36000000"/>

    <workflow config="">

        <workstage name="init">
            <work name="init" type="init" workers="1"
                division="container" totalOps="1" config="containers=r
                     (1,2)">
            </work>
        </workstage>

        <workstage name="prepare">
            <work name="prepare" type="normal" workers="1" totalOps="10">
                <operation type="filewrite" division="none"
                    config="containers=c(1);fileselection=s;files=/tmp/
                         testfiles/A-prepare"/>
            </work>
        </workstage>

        <workstage name="normal">
            <work name="normal" type="normal" workers="1"
                division="none" totalOps="100">
                <operation type="read" ratio="40" division="none"
                    config="containers=c(1);objects=u(1,10)" id="none"/>
                <operation type="filewrite" ratio="60" division="none"
                    config="containers=c(2);fileselection=s;files=/tmp/
                         testfiles/A"/>
            </work>
        </workstage>

        <workstage name="cleanup">
            <work name="cleanup" type="cleanup" workers="1"
                division="object" totalOps="1" config="containers=r(1,2);
                     objects=r(1,50);">
            </work>
        </workstage>

        <workstage name="dispose">
            <work name="dispose" type="dispose" workers="1"
                division="container" totalOps="1" config="containers=r
                     (1,2)">
            </work>
        </workstage>

    </workflow>
</workload>
```

A.2 Workload B (Traveling User)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<workload name="WorkloadB" description="Traveling User" config="">

    <auth type="swauth" config="timeout=36000000;username=test:tester;
         password=testing;auth_url=http://olav-openstack-test-server:8080/
         auth/v1.0"/>
    <storage type="swift" config="timeout=36000000"/>

    <workflow config="">

        <workstage name="init">
            <work name="init" type="init" workers="1"
                division="container" totalOps="1" config="containers=r
                     (1,2)">
            </work>
        </workstage>

        <workstage name="prepare">
            <work name="prepare" type="normal" workers="1" totalOps="10">
        <operation type="filewrite" division="none"
                    config="containers=c(1);fileselection=s;files=/tmp/
                         testfiles/B-prepare"/>
            </work>
        </workstage>

        <workstage name="normal">
            <work name="normal" type="normal" workers="1"
                division="none" totalOps="100">
                <operation type="read" ratio="90" division="none"
                    config="containers=c(1);objects=u(1,10)" id="none"/>
                <operation type="filewrite" ratio="10" division="none"
                    config="containers=c(2);fileselection=s;files=/tmp/
                         testfiles/B"/>
            </work>
        </workstage>

        <workstage name="cleanup">
            <work name="cleanup" type="cleanup" workers="1"
                division="object" totalOps="1" config="containers=r(1,2);
                     objects=r(1,50);">
            </work>
        </workstage>

        <workstage name="dispose">
            <work name="dispose" type="dispose" workers="1"
                division="container" totalOps="1" config="containers=r
                     (1,2)">
            </work>
        </workstage>

    </workflow>
</workload>
```

A.3 Workload C (Email Backup)

```
<?xml version="1.0" encoding="UTF-8"?>
<workload name="WorkloadC" description="Email Backup" config="">

    <auth type="swauth" config="timeout=36000000;username=test:tester;
         password=testing;auth_url=http://olav-openstack-test-server:8080/
         auth/v1.0"/>
    <storage type="swift" config="timeout=36000000"/>

    <workflow config="">

        <workstage name="init">
            <work name="init" type="init" workers="1"
                division="container" totalOps="1" config="containers=r
                     (1,2)">
            </work>
        </workstage>

        <workstage name="prepare">
            <work name="prepare" type="normal" workers="1" totalOps="10">
        <operation type="filewrite" division="none"
                    config="containers=c(1);fileselection=s;files=/tmp/
                         testfiles/C-prepare"/>
            </work>
        </workstage>

        <workstage name="normal">
            <work name="normal" type="normal" workers="1"
                division="none" totalOps="100">
                <operation type="read" ratio="10" division="none"
                    config="containers=c(1);objects=u(1,10);" id="none"/>
                <operation type="filewrite" ratio="90" division="none"
                    config="containers=c(2);fileselection=s;files=/tmp/
                         testfiles/C"/>
            </work>
        </workstage>

        <workstage name="cleanup">
            <work name="cleanup" type="cleanup" workers="1"
                division="object" totalOps="1" config="containers=r(1,2);
                     objects=r(1,50);">
            </work>
        </workstage>

        <workstage name="dispose">
            <work name="dispose" type="dispose" workers="1"
                division="container" totalOps="1" config="containers=r
                     (1,2);">
            </work>
        </workstage>

    </workflow>
</workload>
```

A.4 Workload D (Video Sharing)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<workload name="WorkloadD" description="Code Sharing" config="">

    <auth type="swauth" config="timeout=36000000;username=test:tester;
         password=testing;auth_url=http://olav-openstack-test-server:8080/
         auth/v1.0"/>
    <storage type="swift" config="timeout=36000000"/>

    <workflow config="">

        <workstage name="init">
            <work name="init" type="init" workers="1"
                division="container" totalOps="1" config="containers=r
                     (1,2)">
            </work>
        </workstage>

        <workstage name="prepare">
            <work name="prepare" type="normal" workers="1" totalOps="10">
        <operation type="filewrite" division="none"
                    config="containers=c(1);fileselection=s;files=/tmp/
                         testfiles/D-prepare"/>
            </work>
        </workstage>

        <workstage name="normal">
            <work name="normal" type="normal" workers="1"
                division="none" totalOps="100">
                <operation type="read" ratio="80" division="none"
                    config="containers=c(1);objects=u(1,10)" id="none"/>
                <operation type="filewrite" ratio="20" division="none"
                    config="containers=c(2);fileselection=s;files=/tmp/
                         testfiles/D"/>
            </work>
        </workstage>

        <workstage name="cleanup">
            <work name="cleanup" type="cleanup" workers="1"
                division="object" totalOps="1" config="containers=r(1,2);
                     objects=r(1,50)">
            </work>
        </workstage>

        <workstage name="dispose">
            <work name="dispose" type="dispose" workers="1"
                division="container" totalOps="1" config="containers=r
                     (1,2)">
            </work>
        </workstage>

    </workflow>
</workload>
```

A.5 Workload E (E-Government)

```
<?xml version="1.0" encoding="UTF-8"?>
<workload name="WorkloadE" description="E-Government" config="">

    <auth type="swauth" config="timeout=36000000;username=test:tester;
         password=testing;auth_url=http://olav-openstack-test-server:8080/
         auth/v1.0"/>
    <storage type="swift" config="timeout=36000000"/>

    <workflow config="">

        <workstage name="init">
            <work name="init" type="init" workers="1"
                division="container" totalOps="1" config="containers=r
                     (1,2)">
            </work>
        </workstage>

        <workstage name="prepare">
            <work name="prepare" type="normal" workers="1" totalOps="10">
        <operation type="filewrite" division="none"
                    config="containers=c(1);fileselection=s;files=/tmp/
                         testfiles/E-prepare"/>
            </work>
        </workstage>

        <workstage name="normal1">
            <work name="normal1" type="normal" workers="1"
                division="none" totalOps="20">
                <operation type="read" ratio="95" division="none"
                    config="containers=c(1);objects=u(1,10)" id="none"/>
                <operation type="filewrite" ratio="5" division="none"
                    config="containers=c(2);fileselection=s;files=/tmp/
                         testfiles/E"/>
            </work>
        </workstage>

        <workstage name="normal2">
            <work name="normal2" type="normal" workers="1"
                division="none" totalOps="20">
                <operation type="read" ratio="5" division="none"
                    config="containers=c(1);objects=u(1,10)" id="none"/>
                <operation type="filewrite" ratio="95" division="none"
                    config="containers=c(2);fileselection=s;files=/tmp/
                         testfiles/E"/>
            </work>
        </workstage>

        <workstage name="normal3">
            <work name="normal3" type="normal" workers="1"
                division="none" totalOps="20">
                <operation type="read" ratio="95" division="none"
                    config="containers=c(1);objects=u(1,10)" id="none"/>
                <operation type="filewrite" ratio="5" division="none"
                    config="containers=c(2);fileselection=s;files=/tmp/
                         testfiles/E"/>
            </work>
        </workstage>
        ...
```

```
                ...
        <workstage name="normal4">
            <work name="normal4" type="normal" workers="1"
                division="none" totalOps="20">
                <operation type="read" ratio="5" division="none"
                    config="containers=c(1);objects=u(1,10)" id="none"/>
                <operation type="filewrite" ratio="95" division="none"
                    config="containers=c(2);fileselection=s;files=/tmp/
                        testfiles/E"/>
            </work>
        </workstage>

        <workstage name="normal5">
            <work name="normal5" type="normal" workers="1"
                division="none" totalOps="20">
                <operation type="read" ratio="95" division="none"
                    config="containers=c(1);objects=u(1,10)" id="none"/>
                <operation type="filewrite" ratio="5" division="none"
                    config="containers=c(2);fileselection=s;files=/tmp/
                        testfiles/E"/>
            </work>
        </workstage>

        <workstage name="cleanup">
            <work name="cleanup" type="cleanup" workers="1"
                division="object" totalOps="1" config="containers=r(1,2);
                    objects=r(1,50)">
            </work>
        </workstage>

        <workstage name="dispose">
            <work name="dispose" type="dispose" workers="1"
                division="container" totalOps="1" config="containers=r
                    (1,2)">
            </work>
        </workstage>

    </workflow>
</workload>
```

Appendix B Cost Estimates

This appendix contains the data gathered when estimating the costs of running tests on the IBM cloud. These results were gathered by running tests locally (i.e using the local filesystem instead of a cloud storage service) and recording the number of read/write operations as well as the number of bytes that were read or written. Estimated prices were then calculated using pricing information from Table 4.5.

The appendix is organized into five sections, where each section contains the data for one of the five workloads presented in subsection 4.3.1.

B.1 Workload A (Home User)

	Block Size						
	4 KB	16 KB	64 KB	256 KB	1 MB	4 MB	16 MB
Read operations	29319	12843	1421	603	121	56	38
Write operations	11770	2962	777	242	108	78	77
MBs downloaded	114.25	200.55	88.8	150.74	121	224	608
MBs uploaded	45.86	46.25	48.56	60.5	108	312	1232
Estimated cost	$0.08	$0.04	$0.01	$0.01	$0.01	$0.02	$0.05

Table B.1: Data used to estimate the cost of running Workload A for a range of block sizes without ORAM. Cost is given in USD.

B.2 Workload B (Traveling User)

	Block Size						
	4 KB	16 KB	64 KB	256 KB	1 MB	4 MB	16 MB
Read operations	10729	2410	937	279	119	89	94
Write operations	1972	499	133	46	25	21	16
MBs downloaded	41.81	37.63	58.55	69.75	119	356	1504
MBs uploaded	7.68	7.79	8.31	11.5	25	84	256
Estimated cost	$0.02	$0.01	$0.01	$0.01	$0.01	$0.03	$0.13

Table B.2: Data used to estimate the cost of running Workload B for a range of block sizes without ORAM. Cost is given in USD.

B.3 Workload C (Email Backup)

	Block Size						
	4 KB	16 KB	64 KB	256 KB	1 MB	4 MB	16 MB
Read operations	51956	11219	3759	1003	145	19	13
Write operations	424157	107946	25919	6551	1832	510	162
MBs downloaded	202.46	175.19	234.9	250.74	145	76	208
MBs uploaded	1652.82	1685.63	1619.69	1637.69	1831.98	2040	2592
Estimated cost	$2.16	$0.56	$0.15	$0.06	$0.02	$0.01	$0.02

Table B.3: Data used to estimate the cost of running Workload C for a range of block sizes without ORAM. Cost is given in USD.

B.4 Workload D (Code Distribution)

	Block Size						
	4 KB	16 KB	64 KB	256 KB	1 MB	4 MB	16 MB
Read operations	6271	643	346	129	88	78	83
Write operations	610	172	70	37	40	32	27
MBs downloaded	24.44	10.04	21.62	32.25	88	312	1328
MBs uploaded	2.38	2.69	4.37	9.25	40	128	432
Estimated cost	$0.0077	$0.0020	$0.0024	$0.0031	$0.0080	$0.0276	$0.1169

Table B.4: Data used to estimate the cost of running Workload D for a range of block sizes without ORAM. Cost is given in USD.

B.5 Workload E (E-Government)

	Block Size						
	4 KB	16 KB	64 KB	256 KB	1 MB	4 MB	16 MB
Read operations	55753	12785	4124	979	231	80	59
Write operations	63823	16610	3855	1014	280	98	51
MBs downloaded	217.25	199.64	257.71	244.74	231	320	944
MBs uploaded	248.7	259.37	240.9	253.49	280	392	816
Estimated cost	$0.36	$0.11	$0.04	$0.03	$0.02	$0.03	$0.08

Table B.5: Data used to estimate the cost of running Workload E for a range of block sizes without ORAM. Cost is given in USD.

Appendix C Full results

This appendix contains the data that was gathered from all experiments.The data is presented in a set of tables. Each table corresponds to a workload and an ORAM scheme (e.g Workload A, RingORAM). The tables are divided into two sections. The first section contains data that was measured during the test, while the second section contains calculated metrics.

C.1 Workload A (Home User)

		Block Size		
		64 KB	256 KB	1 MB
No ORAM (with blocks)	Download operations	1824	162	176
	Upload operations	2794	744	231
	MBs downloaded	113.98	40.5	176
	MBs uploaded	174.6	185.99	231
	Remote storage (MB)	174.6	186	231
	Peak local storage (MB)	0.1	0.2	1
	Runtime (seconds)	250	75	55
	Average CPU usage (%)	1.67	3.46	5.3
	Peak CPU usage (%)	11.2	9.1	8.3
	Average RAM usage (MB)	159.29	233.88	175.21
	Peak RAM usage (MB)	411.42	363.04	437.58
	Read bandwidth (KB/s)	473.24	482.87	2882.79
	Write bandwidth (KB/s)	600.47	2029.09	2871.32
	Avg. response time read (ms)	1625.4	262.03	403.56
	Avg. response time write (ms)	3066.68	1004.16	600.66
	Cost (USD)	$0.02	$0.01	$0.02
	Bandwidth overhead	1.44	1.13	2.03
	Relative response time	9.43	2.54	2.02
	Relative slowdown	25	7.5	5.5
	Cost multiplier	2	1	2
	Outsource ratio	51200	25600	5120

		Block Size		
		64 KB	256 KB	1 MB
ObliviStore	Download operations	33270	10927	2541
	Upload operations	340270	93664	23270
	MBs downloaded	2079.38	2731.75	2541
	MBs uploaded	3705.63	5237.5	4774
	Remote storage (MB)	19362.6	20336	20736
	Peak local storage (MB)	80.9	72.8	106
	Runtime (seconds)	100	85	90
	Average CPU usage (%)	57.02	67.31	62.38
	Peak CPU usage (%)	81.3	89.8	88.8
	Average RAM usage (MB)	3287.1	2464.11	2149.44
	Peak RAM usage (MB)	7369.36	5938.9	4688.51
	Read bandwidth (KB/s)	625.14	1796.14	1308.47
	Write bandwidth (KB/s)	1527.28	1405.58	1655.7
	Avg. response time read (ms)	902.98	545.7	907.41
	Avg. response time write (ms)	1028.81	1180.89	900.36
	Cost (USD)	$1.92	$0.72	$0.36
	Bandwidth overhead	28.83	39.71	36.45
	Relative response time	3.88	3.47	3.63
	Relative slowdown	10	8.5	9
	Cost multiplier	192	72	36
	Outsource ratio	63.29	70.33	48.3

		Block Size		
		64 KB	256 KB	1 MB
CURIOUS	Download operations	29855	7300	1196
	Upload operations	70553	26184	4375
	MBs downloaded	9328.78	9124.78	5979.96
	MBs uploaded	9701.96	9489.78	6278.96
	Remote storage (MB)	11224.3	22144.5	14698.9
	Peak local storage (MB)	30.2	44	96.7
	Runtime (seconds)	170	180	645
	Average CPU usage (%)	70.26	64.26	32.64
	Peak CPU usage (%)	91.2	94.8	89.8
	Average RAM usage (MB)	1289.09	4181.22	3110.64
	Peak RAM usage (MB)	3095.35	9905.17	7496.13
	Read bandwidth (KB/s)	1233	1019.88	54.98
	Write bandwidth (KB/s)	893.89	830.58	231.99
	Avg. response time read (ms)	2896.32	1936.51	1462.53
	Avg. response time write (ms)	1123.84	1661.49	10470.76
	Cost (USD)	$1.19	$0.97	$0.55
	Bandwidth overhead	94.83	92.75	61.08
	Relative response time	8.08	7.23	23.98
	Relative slowdown	17	18	64.5
	Cost multiplier	119	97	55
	Outsource ratio	169.54	116.36	52.95

		Block Size		
		64 KB	256 KB	1 MB
RingORAM	Download operations	185366	50708	8584
	Upload operations	56108	15143	2716
	MBs downloaded	9250.57	9876.44	5969.76
	MBs uploaded	6332.93	6727.37	3841.25
	Remote storage (MB)	24568.7	24551.7	24479.9
	Peak local storage (MB)	7.42	28.79	115.01
	Runtime (seconds)	14403	3529	871
	Average CPU usage (%)	0.51	1.57	3.63
	Peak CPU usage (%)	10.4	15.7	23.2
	Average RAM usage (MB)	359.09	334.3	841.49
	Peak RAM usage (MB)	588.83	586.93	1485.45
	Read bandwidth (KB/s)	6.32	43.62	8.66
	Write bandwidth (KB/s)	10.39	42.26	171.43
	Avg. response time read (ms)	126654.42	38772.4	2069.4
	Avg. response time write (ms)	154615.02	32706.52	14095.96
	Cost (USD)	$1.2	$0.98	$0.56
	Bandwidth overhead	77.65	82.73	48.89
	Relative response time	565.25	143.65	32.49
	Relative slowdown	1440.3	352.9	87.1
	Cost multiplier	120	98	56
	Outsource ratio	689.85	177.85	44.52

C.2 Workload B (Traveling User)

		Block Size		
		16 KB	64 KB	256 KB
No ORAM (with blocks)	Download operations	2196	627	228
	Upload operations	471	144	52
	MBs downloaded	34.29	39.18	57
	MBs uploaded	7.35	9	13
	Remote storage (MB)	7.4	9	13
	Peak local storage (MB)	0.02	0.1	0.2
	Runtime (seconds)	85	35	20
	Average CPU usage (%)	1.68	2.92	4.59
	Peak CPU usage (%)	4.6	5.6	7.9
	Average RAM usage (MB)	88.98	91.71	90.74
	Peak RAM usage (MB)	165.39	158.91	161.1
	Read bandwidth (KB/s)	415.03	1177.78	2320.73
	Write bandwidth (KB/s)	31.02	106.13	177.79
	Avg. response time read (ms)	766.96	301.25	171.02
	Avg. response time write (ms)	2044.67	373.38	194.4
	Cost (USD)	$0.01	$0	$0.01
	Bandwidth overhead	4.48	5.18	7.53
	Relative response time	6.98	1.68	0.91
	Relative slowdown	17	7	4
	Cost multiplier	33.54	14.81	33.54
	Outsource ratio	327680	51200	25600

		Block Size		
		16 KB	64 KB	256 KB
ObliviStore	Download operations	44235	8496	2327
	Upload operations	1173332	294284	75494
	MBs downloaded	691.17	531	581.75
	MBs uploaded	1243.44	861.63	936.5
	Remote storage (MB)	17735.8	18121.1	18558
	Peak local storage (MB)	12.3	6.9	8.1
	Runtime (seconds)	111	30	265
	Average CPU usage (%)	47.65	53.94	28.31
	Peak CPU usage (%)	62.5	69.4	71.1
	Average RAM usage (MB)	1231.4	1431.42	404.27
	Peak RAM usage (MB)	2210.92	2570.49	1233.51
	Read bandwidth (KB/s)	478.93	1876.76	138.96
	Write bandwidth (KB/s)	27.61	119.55	11.45
	Avg. response time read (ms)	1125.99	247.51	2976.41
	Avg. response time write (ms)	518.43	300.42	263
	Cost (USD)	$5.99	$1.55	$0.45
	Bandwidth overhead	208.01	149.73	163.24
	Relative response time	4.08	1.36	8.04
	Relative slowdown	22.2	6	53
	Cost multiplier	20092.94	5199.34	1509.49
	Outsource ratio	416.26	742.03	632.1

		Block Size		
		16 KB	64 KB	256 KB
CURIOUS	Download operations	19770	5250	1975
	Upload operations	127533	41027	19794
	MBs downloaded	1543.93	1640.46	2468.69
	MBs uploaded	1595.41	1706.09	2567.44
	Remote storage (MB)	8209.9	10916.8	21878.2
	Peak local storage (MB)	15.4	8.7	9.3
	Runtime (seconds)	55	50	95
	Average CPU usage (%)	52.26	54.34	46.9
	Peak CPU usage (%)	70.1	84.6	74.8
	Average RAM usage (MB)	935.39	664.42	1359.56
	Peak RAM usage (MB)	2051.12	1296.9	3328.1
	Read bandwidth (KB/s)	813.98	1147.85	794.22
	Write bandwidth (KB/s)	55.07	62.32	31.93
	Avg. response time read (ms)	583.99	498.79	959.2
	Avg. response time write (ms)	337.62	337.89	690.25
	Cost (USD)	$0.79	$0.36	$0.33
	Bandwidth overhead	337.54	359.81	541.47
	Relative response time	2.29	2.08	4.1
	Relative slowdown	11	10	19
	Cost multiplier	2649.99	1207.59	1106.96
	Outsource ratio	332.47	588.51	550.54

		Block Size		
		16 KB	64 KB	256 KB
RingORAM	Download operations	158652	31568	11956
	Upload operations	73764	16337	5125
	MBs downloaded	2017.18	1521.75	2184.99
	MBs uploaded	1358.06	1024.59	1441.74
	Remote storage (MB)	24569.3	24568.7	24551.7
	Peak local storage (MB)	2.73	7.42	28.79
	Runtime (seconds)	8121	1767	792
	Average CPU usage (%)	0.4	0.65	1.65
	Peak CPU usage (%)	5.9	12.9	19.1
	Average RAM usage (MB)	167.08	238.46	257.13
	Peak RAM usage (MB)	273.29	351.65	458.93
	Read bandwidth (KB/s)	5.08	18.96	65.86
	Write bandwidth (KB/s)	0.37	1.71	3.87
	Avg. response time read (ms)	81832.75	17493.64	8452.73
	Avg. response time write (ms)	73670	18891.33	4481.93
	Cost (USD)	$0.63	$0.24	$0.25
	Bandwidth overhead	362.9	273.78	389.94
	Relative response time	386.13	90.35	32.12
	Relative slowdown	1624.2	353.4	158.4
	Cost multiplier	2113.28	805.06	838.6
	Outsource ratio	1872.46	689.85	177.85

C.3 Workload C (Email Backup)

		Block Size		
		1 MB	4 MB	16 MB
No ORAM (with blocks)	Download operations	73	57	12
	Upload operations	1925	473	166
	MBs downloaded	73	228	192
	MBs uploaded	1924.98	1892	2656
	Remote storage (MB)	1925	1892	2656
	Peak local storage (MB)	1	4	16
	Runtime (seconds)	315	546	475
	Average CPU usage (%)	5.17	2.62	3.62
	Peak CPU usage (%)	20.2	17	15
	Average RAM usage (MB)	197.86	174.37	292.21
	Peak RAM usage (MB)	291.47	284.88	730.88
	Read bandwidth (KB/s)	234.77	382.81	293.51
	Write bandwidth (KB/s)	5550.7	2900.91	3533.25
	Avg. response time read (ms)	2163.33	1178.58	1076.29
	Avg. response time write (ms)	3156.04	6033.31	4989.44
	Cost (USD)	$0.02	$0.02	$0.02
	Bandwidth overhead	1.08	1.14	1.54
	Relative response time	4.6	6.24	5.25
	Relative slowdown	4.5	7.8	6.79
	Cost multiplier	1	1	1
	Outsource ratio	5120	1280	320

		Block Size		
		1 MB	4 MB	16 MB
ObliviStore	Download operations	14813	3566	1012
	Upload operations	48878	13296	4090
	MBs downloaded	14813	14264	16192
	MBs uploaded	30484	31920	40672
	Remote storage (MB)	23328	25056	28224
	Peak local storage (MB)	123.6	168	368
	Runtime (seconds)	360	416	1011
	Average CPU usage (%)	74.03	73.39	52.69
	Peak CPU usage (%)	89.7	96.4	97.9
	Average RAM usage (MB)	3482.48	6173.45	8575.68
	Peak RAM usage (MB)	7586.21	11211.36	15426.42
	Read bandwidth (KB/s)	394.88	335.38	164.96
	Write bandwidth (KB/s)	4583.53	3938.15	1589.38
	Avg. response time read (ms)	4076.38	4078.56	5895.9
	Avg. response time write (ms)	3541.77	4131.66	10574.9
	Cost (USD)	$1.56	$1.33	$1.46
	Bandwidth overhead	24.46	24.94	30.7
	Relative response time	6.59	7.1	14.25
	Relative slowdown	5.14	5.94	14.44
	Cost multiplier	78	66.5	73
	Outsource ratio	41.42	30.48	13.91

		Block Size		
		1 MB	4 MB	16 MB
CURIOUS	Download operations	7844	2088	528
	Upload operations	12685	4050	938
	MBs downloaded	39219.76	41759.94	42239.98
	MBs uploaded	41180.76	43847.94	45055.98
	Remote storage (MB)	16360.9	30888	21536
	Peak local storage (MB)	201.2	285.4	416
	Runtime (seconds)	846	1552	1488
	Average CPU usage (%)	62.22	45.48	45.08
	Peak CPU usage (%)	96	87.5	88.4
	Average RAM usage (MB)	5584.94	6195.38	6313.62
	Peak RAM usage (MB)	10572.41	13011.46	13268.1
	Read bandwidth (KB/s)	171.55	133.95	26.08
	Write bandwidth (KB/s)	1926.86	997.83	1166.76
	Avg. response time read (ms)	7358.22	8734.79	7297
	Avg. response time write (ms)	8541.27	16601.44	15107.8
	Cost (USD)	$3.53	$3.8	$3.81
	Bandwidth overhead	43.41	46.22	47.14
	Relative response time	13.75	21.91	19.38
	Relative slowdown	12.09	22.17	21.26
	Cost multiplier	176.5	190	190.5
	Outsource ratio	25.45	17.94	12.31

		Block Size		
		1 MB	4 MB	16 MB
RingORAM	Download operations	61056	12256	2756
	Upload operations	16272	3289	737
	MBs downloaded	46996.72	37473.36	29361.21
	MBs uploaded	31880.88	25345.78	18432.4
	Remote storage (MB)	24479.9	24192	23040
	Peak local storage (MB)	115.01	460	1840
	Runtime (seconds)	19505	8721	2442
	Average CPU usage (%)	1.26	2.03	5.65
	Peak CPU usage (%)	28.8	45.1	63.1
	Average RAM usage (MB)	1863.65	3834.78	9050.96
	Peak RAM usage (MB)	3186.34	9576.39	22250.06
	Read bandwidth (KB/s)	8.71	25.14	62.73
	Write bandwidth (KB/s)	83.34	178.71	665.95
	Avg. response time read (ms)	27176.89	135468.77	71253.56
	Avg. response time write (ms)	211624.88	79975.77	19772.23
	Cost (USD)	$4.25	$3.35	$2.6
	Bandwidth overhead	42.59	33.92	25.81
	Relative response time	206.55	186.35	78.73
	Relative slowdown	278.64	124.59	34.89
	Cost multiplier	212.5	167.5	130
	Outsource ratio	44.52	11.13	2.78

C.4 Workload D (Code Distribution)

		Block Size		
		16 KB	64 KB	256 KB
No ORAM (with blocks)	Download operations	1190	357	159
	Upload operations	170	59	39
	MBs downloaded	18.58	22.31	39.75
	MBs uploaded	2.65	3.69	9.75
	Remote storage (MB)	2.7	3.7	9.7
	Peak local storage (MB)	0.02	0.1	0.2
	Runtime (seconds)	45	20	15
	Average CPU usage (%)	1.75	1.24	1.61
	Peak CPU usage (%)	3.8	3.3	8.6
	Average RAM usage (MB)	84.91	60.74	43.59
	Peak RAM usage (MB)	148.33	78.61	77.64
	Read bandwidth (KB/s)	428.52	1106.15	1695.2
	Write bandwidth (KB/s)	2.16	4.24	6.99
	Avg. response time read (ms)	512.48	179.64	135.04
	Avg. response time write (ms)	121.15	106.33	143.9
	Cost (USD)	$0	$0	$0
	Bandwidth overhead	1.05	1.29	2.46
	Relative response time	1.84	0.83	0.81
	Relative slowdown	9	4	3
	Cost multiplier	1.66	1.35	2.11
	Outsource ratio	327680	51200	25600

		Block Size		
		16 KB	64 KB	256 KB
ObliviStore	Download operations	20255	4733	1441
	Upload operations	1124420	288230	75074
	MBs downloaded	316.48	295.81	360.25
	MBs uploaded	534.34	461.88	555.5
	Remote storage (MB)	17341.8	17927.4	18634.5
	Peak local storage (MB)	10.6	4.9	7.4
	Runtime (seconds)	55	20	260
	Average CPU usage (%)	45.47	0.22	25.73
	Peak CPU usage (%)	73.1	1.3	37.9
	Average RAM usage (MB)	757.82	2979.32	439.25
	Peak RAM usage (MB)	1377.8	3068.37	678.05
	Read bandwidth (KB/s)	451.57	1166.07	39.39
	Write bandwidth (KB/s)	2.69	8.23	0.42
	Avg. response time read (ms)	516.71	177.93	209.66
	Avg. response time write (ms)	492.96	123.42	10560.39
	Cost (USD)	$5.7	$1.49	$0.42
	Bandwidth overhead	42.22	37.6	45.45
	Relative response time	2.94	0.88	31.34
	Relative slowdown	11	4	52
	Cost multiplier	3205.86	838.02	236.22
	Outsource ratio	483.02	1044.9	691.89

		Block Size		
		16 KB	64 KB	256 KB
CURIOUS	Download operations	6552	2590	950
	Upload operations	216216	37835	18564
	MBs downloaded	511.68	809.3	1187.47
	MBs uploaded	528.74	841.67	1234.97
	Remote storage (MB)	16305.4	10883.5	21827
	Peak local storage (MB)	14.1	5.6	8.5
	Runtime (seconds)	20	1701	300
	Average CPU usage (%)	17.61	25.79	27.45
	Peak CPU usage (%)	67.9	66.9	68.1
	Average RAM usage (MB)	898.86	319.36	1073.9
	Peak RAM usage (MB)	1412.06	622.39	1935.98
	Read bandwidth (KB/s)	832.69	15.26	73.22
	Write bandwidth (KB/s)	5.1	0.03	0.45
	Avg. response time read (ms)	197.44	18428.24	3766.45
	Avg. response time write (ms)	57.78	174.12	569.72
	Cost (USD)	$1.14	$0.27	$0.21
	Bandwidth overhead	51.63	81.93	120.22
	Relative response time	0.74	54.14	12.62
	Relative slowdown	4	340.2	60
	Cost multiplier	641.17	151.86	118.11
	Outsource ratio	363.12	914.29	602.35

		Block Size		
		16 KB	64 KB	256 KB
RingORAM	Download operations	69842	10204	7716
	Upload operations	50816	10831	4035
	MBs downloaded	889.8	522.02	1440.63
	MBs uploaded	599.65	361.48	961.12
	Remote storage (MB)	24569.3	24568.7	24551.7
	Peak local storage (MB)	2.73	7.42	28.79
	Runtime (seconds)	3818	521	505
	Average CPU usage (%)	0.29	0.69	1.68
	Peak CPU usage (%)	5.6	12.7	15.9
	Average RAM usage (MB)	141.06	132.34	289.41
	Peak RAM usage (MB)	235.55	212.28	627.53
	Read bandwidth (KB/s)	4.88	14.58	50.68
	Write bandwidth (KB/s)	0.03	0.17	0.14
	Avg. response time read (ms)	47746.94	5629.59	5280.79
	Avg. response time write (ms)	4206.59	3285.21	3553.93
	Cost (USD)	$0.39	$0.14	$0.18
	Bandwidth overhead	73.92	43.85	119.19
	Relative response time	151.2	25.95	25.71
	Relative slowdown	763.6	104.2	101
	Cost multiplier	219.35	78.74	101.24
	Outsource ratio	1872.46	689.85	177.85

C.5 Workload E (E-Government)

		Block Size		
		256 KB	1 MB	4 MB
No ORAM (with blocks)	Download operations	917	248	92
	Upload operations	1030	265	84
	MBs downloaded	229.24	248	368
	MBs uploaded	257.49	265	336
	Remote storage (MB)	257.5	265	336
	Peak local storage (MB)	0.2	1	4
	Runtime (seconds)	167	86	67
	Average CPU usage (%)	3.2	2.99	3.79
	Peak CPU usage (%)	7.3	11.2	12.4
	Average RAM usage (MB)	99.61	78.01	97.81
	Peak RAM usage (MB)	188.52	183.13	178.65
	Read bandwidth (KB/s)	2428.36	5464.21	8449.3
	Write bandwidth (KB/s)	1194.46	2415.05	3071.77
	Avg. response time read (ms)	853.49	358.98	249.5
	Avg. response time write (ms)	2569.45	970.63	923.87
	Cost (USD)	$0.03	$0.02	$0.03
	Bandwidth overhead	1.05	1.11	1.52
	Relative response time	3.55	1.38	1.22
	Relative slowdown	3.55	1.83	1.43
	Cost multiplier	1.5	1	1.5
	Outsource ratio	25600	5120	1280

		Block Size		
		256 KB	1 MB	4 MB
ObliviStore	Download operations	16608	4123	1237
	Upload operations	104606	27494	7920
	MBs downloaded	4152	4123	4948
	MBs uploaded	8153.5	8456	11032
	Remote storage (MB)	20992	21856	24032
	Peak local storage (MB)	38.8	61	108
	Runtime (seconds)	137	146	207
	Average CPU usage (%)	60.41	58.75	60.37
	Peak CPU usage (%)	87.1	87.7	94.7
	Average RAM usage (MB)	1478.59	2960.67	5997.44
	Peak RAM usage (MB)	3259.08	6012.81	10398.58
	Read bandwidth (KB/s)	2466.17	1722.96	1432.16
	Write bandwidth (KB/s)	1625.5	1914.65	1251.12
	Avg. response time read (ms)	951.77	1255.88	1651.76
	Avg. response time write (ms)	1409.14	1164.62	1860.97
	Cost (USD)	$0.91	$0.51	$0.49
	Bandwidth overhead	26.65	27.24	34.61
	Relative response time	2.45	2.51	3.64
	Relative slowdown	2.91	3.11	4.4
	Cost multiplier	45.5	25.5	24.5
	Outsource ratio	131.96	83.93	47.41

		Block Size		
		256 KB	1 MB	4 MB
CURIOUS	Download operations	9845	2040	668
	Upload operations	29238	5430	2275
	MBs downloaded	12305.95	10199.94	13359.98
	MBs uploaded	12798.2	10709.94	14027.98
	Remote storage (MB)	22271.7	14909.9	29468
	Peak local storage (MB)	43	73	152
	Runtime (seconds)	292	1002	407
	Average CPU usage (%)	57.5	33.73	46.72
	Peak CPU usage (%)	90.2	90.2	80
	Average RAM usage (MB)	3180.95	1811.72	4644.34
	Peak RAM usage (MB)	8372.2	7113.69	8983.32
	Read bandwidth (KB/s)	1037.83	651.8	566.16
	Write bandwidth (KB/s)	690.32	732.56	510.13
	Avg. response time read (ms)	2212.26	9229.17	3089.42
	Avg. response time write (ms)	3140.3	3240.45	4967.29
	Cost (USD)	$1.24	$0.93	$1.26
	Bandwidth overhead	54.37	45.29	59.32
	Relative response time	5.55	12.92	8.35
	Relative slowdown	6.21	21.32	8.66
	Cost multiplier	62	46.5	63
	Outsource ratio	119.07	70.14	33.68

		Block Size		
		256 KB	1 MB	4 MB
RingORAM	Download operations	72822	15064	3928
	Upload operations	20856	4396	1135
	MBs downloaded	14204.86	11412.59	10805.72
	MBs uploaded	9682.59	7682.19	6912.57
	Remote storage (MB)	24551.7	24479.9	24192
	Peak local storage (MB)	28.79	115.01	460
	Runtime (seconds)	5320	1352	877
	Average CPU usage (%)	1.48	3.92	5.74
	Peak CPU usage (%)	15.9	20.4	33.8
	Average RAM usage (MB)	376	996.25	2432.51
	Peak RAM usage (MB)	650.87	2091.67	5724.32
	Read bandwidth (KB/s)	52.48	168.43	512.64
	Write bandwidth (KB/s)	35.2	147.08	174.26
	Avg. response time read (ms)	59032.98	10186.55	5000.86
	Avg. response time write (ms)	69472.44	16652.46	10799.73
	Cost (USD)	$1.4	$1.06	$0.97
	Bandwidth overhead	51.74	41.36	38.37
	Relative response time	133.15	27.81	16.37
	Relative slowdown	113.19	28.77	18.66
	Cost multiplier	70	53	48.5
	Outsource ratio	177.85	44.52	11.13